FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Secure Coordination through Fine-Grained Access Control for Space-Based Computing Middleware

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der Technischen Wissenschaften

eingereicht von

DI Stefan Craß
Matrikelnummer 00325656

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Ao. Univ. Prof. DI Dr. Eva Maria Kühn

Diese Dissertation haben begutachtet:

René Mayrhofer	Stefanie Rinderle-Ma

Wien, 7. Mai 2020

Stefan Craß

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Secure Coordination through Fine-Grained Access Control for Space-Based Computing Middleware

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

DI Stefan Craß
Registration Number 00325656

to the Faculty of Informatics
at the TU Wien

Advisor: Ao. Univ. Prof. DI Dr. Eva Maria Kühn

The dissertation has been reviewed by:

René Mayrhofer	Stefanie Rinderle-Ma

Vienna, 7th May, 2020

Stefan Craß

Erklärung zur Verfassung der Arbeit

DI Stefan Craß
Gassergasse 32/16, 1050 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 7. Mai 2020

Stefan Craß

Acknowledgements

This thesis would not have been possible without the help of many supporters.

First, I want to thank my supervisor Eva Maria Kühn for her continuous support. She was always available for fruitful technical discussions and helpful feedback on my work. With gratitude I look back on my long-standing occupation as her research assistant in the Space-Based Computing group, which gave me the opportunity to work on many interesting projects and topics.

Additionally, I would like to thank all colleagues and students that supported my research with their work on middleware concepts, prototype implementations, and use cases. A special thanks goes to Geri Joskowicz, who provided highly valuable comments on early drafts of this thesis.

Finally, I also want to thank my family and my girlfriend Sara for their encouragement and patience.

Kurzfassung

Die Entwicklung verteilter Systeme mit unterschiedlichen Akteuren und veränderlichen Anforderungen ist eine hochkomplexe Aufgabe, die durch den Einsatz von Middleware mit geeigneten Koordinationskonzepten vereinfacht werden kann. Allerdings müssen in offenen Umgebungen wie dem Internet auch Sicherheit und Vertrauen zwischen den Beteiligten berücksichtigt werden. Jeder Teilnehmer muss in der Lage sein, den Zugriff auf eigene Daten und Services auf flexible Weise zu schützen. Das gilt auch für Space-basierte Middleware, die datengetriebene Koordination zwischen autonomen Prozessen durch entkoppelte Kommunikation über geteilte Datenräume („Spaces“) ermöglicht.

Diese Dissertation strebt daher die Integration von Space-basierter Koordination mit entsprechenden Sicherheitstechnologien an, wobei ein neuartiges Autorisierungs-Konzept definiert wird, das etablierte Grundsätze der Zugriffskontrolle an die charakteristischen Eigenschaften von Space-basierter Middleware anpasst. Das Konzept basiert auf einfachen, aber ausdrucksstarken Autorisierungsregeln, die Operationen auf bestimmten Space-Partitionen einschränken, wodurch eine feingranulare Zugriffskontrolle ermöglicht wird. Berechtigungen können dabei von authentifizierten Subjektattributen, Inhalten der involvierten Objekte sowie zusätzlichen Kontextinformationen abhängen. Administratoren sind dadurch in der Lage, jedem Teilnehmer nur jene Berechtigungen zu erteilen, die für geplante Interaktionen auch tatsächlich notwendig sind. Dieser Ansatz wird anhand von Zugriffskontrollmodellen für zwei verwandte Middleware-Technologien präsentiert, die verschiedene Aspekte von Space-basierter Koordination abdecken. XVSM bietet konfigurierbare Sub-Spaces mit erweiterbaren Query-Funktionen, während das Peer-Modell eine hierarchische Space-Struktur mit anpassbarer Koordinationslogik für die bedingte Ausführung von Message-Routing und Service-Aufrufen unterstützt. Durch die Verwendung intrinsischer Koordinationsmechanismen der jeweiligen Middleware können Autorisierungs-Policies für jeden verteilten Space unabhängig voneinander konfiguriert werden, wobei Administratorenrechte für dynamische Policy-Änderungen auf dieselbe Weise spezifiziert werden wie reguläre Berechtigungen. Die Sicherheit wird zusätzlich verstärkt durch ein mehrschichtiges Schutzmodell, das Berechtigungen auf unterschiedlichen Ebenen erfordert. Zudem ermöglicht ein integriertes Delegations- und Vertrauens-Konzept den Einsatz in offenen Umgebungen ohne vordefinierte Vertrauensverhältnisse.

Um die praktische Einsetzbarkeit zu erreichen, werden die entworfenen Zugriffskontrollmodelle in die jeweiligen Middleware-Architekturen und deren prototypischen Laufzeitsysteme integriert. Durch die Spezifikation von Vorlagen („Patterns“) für sichere Koordination wird Wiederverwendbarkeit gefördert. Diese Patterns bieten generische

Lösungen für verbreitete Koordinationsaufgaben, indem sie die benötigte Koordinationslogik mit geeigneten Autorisierungs-Policies für den Schutz aller beteiligten Spaces kombinieren. Die Umsetzbarkeit des Ansatzes wird durch mehrere Fallstudien belegt, die unterschiedliche Sicherheitsanforderungen und Anwendungsdomänen abdecken.

Abstract

Developing distributed systems with multiple stakeholders and evolving requirements is a highly complex task, which can be simplified by the usage of middleware with suitable coordination abstractions. However, in open environments like the Internet, also security and trust among the participants have to be considered. Each participant must be able to protect access to its own data and services in a flexible way. This also applies to space-based middleware, which enables data-driven coordination among autonomous processes using decoupled communication via shared spaces.

This thesis therefore aims at integrating space-based coordination with security by creating a novel authorization concept that adapts well-established access control principles to the characteristic properties of space-based middleware. The concept relies on simple yet expressive authorization rules that restrict operations on specific space partitions, thus allowing for fine-grained access control. Permissions may depend on authenticated subject attributes, properties of the accessed content, and additional context information. This approach enables administrators to grant each participant only permissions that are actually necessary for planned interactions. It is presented by means of access control models for two related middleware technologies that cover different aspects of space-based coordination. XVSM provides configurable sub-spaces with extensible query features, while the Peer Model supports a hierarchical space structure with customizable coordination logic for conditional message routing and service invocations. Using the intrinsic coordination mechanisms of the respective middleware, authorization policies can be configured independently for each distributed space, whereas administrator privileges for dynamic policy modifications are specified in the same way as regular permissions. Security is further increased by the usage of multiple protection layers, so that permissions need to be acquired at different levels. Due to an integrated delegation and trust concept, the approach is suitable for open environments without fixed trust assumptions.

To enable their practical application, the conceptualized access control models are integrated into the respective middleware architectures and their prototypical runtime implementations. Reusability is promoted via the specification of patterns for secure coordination, which provide generic solutions for common coordination tasks by combining the required coordination logic with suitable authorization policies for protecting all involved spaces. The feasibility of the approach is demonstrated via a series of case studies that cover different security constraints and application domains.

Contents

CHAPTER 1

Introduction

Modern distributed systems often require complex collaboration among autonomous processes. Common challenges include the exchange of data, the invocation of remote services, the distribution of event notifications, synchronization of two or more processes, and consensus about subsequent actions among participants. Besides providing suitable mechanisms for such coordination tasks, a distributed application has to handle communication failures, process crashes, concurrency conflicts, and timeout restrictions in a resilient way. Distributed applications may also require the integration of heterogeneous systems if involved processes run on different types of devices (e.g., servers, personal computers, and mobile phones).

These problems are difficult to solve, which may lead to costly errors in the design and implementation phases. Middleware can help to reduce complexity by providing a proper abstraction level for developers, which enables them to use proven functionality to realize the desired behavior of their distributed application. This approach also masks heterogeneity of devices, as developers only have to interact with uniform middleware services instead of having to deal with low-level functions for multiple platforms. Thus, the development effort and risk can be greatly reduced by the usage of a suitable middleware.

In recent years, several computing trends have emerged that enable new types of distributed applications. Web 2.0 has transformed users from passive consumers to active content creators and site administrators via social networks. Cloud computing supports flexible scaling for enterprises, as computational resources can simply be extended using third-party infrastructure. The Internet of Things (IoT) promises to connect all kinds of "smart" devices to enable new use cases in domains like home automation and intelligent transportation systems. Distributed computing frameworks like BOINC [And04] spread complex scientific computations among users that provide their unused processing resources. Such use cases are characterized by an open environment and the possibility to collaborate on a global scale via the Internet.

In this setting, security becomes a very crucial aspect. Instead of a single organization that needs to protect access to its services and data, many different stakeholders with

diverse security requirements are involved. Each of them may need to provide services and/or data to others in a protected fashion. Distributed applications also do not adhere to a static topology, but involve dynamically joining and leaving participants, which may interact in a peer-to-peer (P2P) way, i.e., via direct communication without central control by a server. Most end users (and also many companies) are not fully aware of the security and privacy considerations induced by such highly collaborative applications. Therefore, security in such *open distributed systems* is often a neglected topic. However, as reinforced by case studies on potentially life-threatening hacks of vehicles [Wri11] or medical implants [AY16], this attitude has to change quickly. In order to propagate the usage of appropriate security mechanisms for distributed applications, the middleware used for communication and coordination should also support suitable security features.

The overall motivation for this thesis is therefore to find a suitable abstraction for such *coordination middleware* that supports the development of secure distributed applications, whereas the focus lies on cross-organizational collaboration and ad-hoc workflows that can be dynamically instantiated after deployment. Users should be able to participate in such scenarios while having full control over who can access their data and their offered services. In contrast to central management by a single administrator, each stakeholder should be entitled to define security policies for their own software components in a decentralized way. An expressive yet usable security model incorporated into a middleware that is designed specifically for complex collaboration within open environments is an important step towards truly secure distributed applications.

Current security measures are mainly targeted on the IT infrastructure of individual organizations, where external access is usually possible only via well-defined services, which may be offered by a Web server. An authentication mechanism identifies accessing parties, whereas an authorization policy checks whether the user is permitted to invoke the service. These services may retrieve data from a backend database, which only permits internal access by the server. In such settings, cross-organizational access to non-public services may be difficult to manage. There has been extensive research on federated security, which has led to security technologies that are feasible for open environments [GGKL89, CO03, OAS05, ITU12], but none of those is integrated with a suitable coordination middleware in a natural way. This thesis aims at closing this gap between coordination and security.

1.1 The Evolution of Middleware

Distributed applications consist of a computational part, which copes with data processing, and a coordination part that enables the communication and cooperation between the involved components [PA98]. Thus, coordination can be defined as the process of building programs by gluing together these active pieces [CG90]. There are many different ways how middleware supports coordination within distributed systems. The most basic middleware paradigm is direct communication, where a client process invokes a service on a remote server using a well-defined interface. Examples include RPC [BN84], RMI [Wal98], and Web services [CDK+02]. Such an interaction style requires that the client

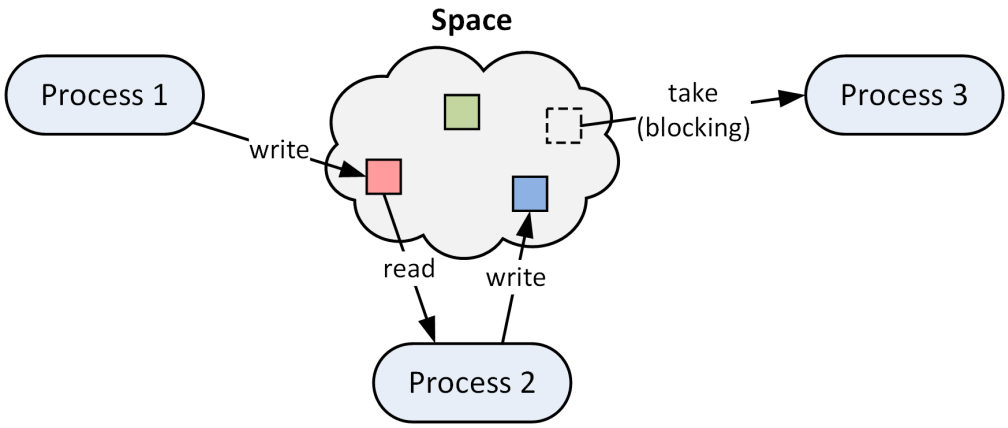

Figure 1.1: Space-based coordination paradigm

knows the server and both participants are online at the same time. For the coordination of multiple autonomous processes, a more decoupled approach is desirable.

Message-oriented middleware like JMS [HBS+13] provides queues that act as intermediary components between message senders and receivers. This facilitates asynchronous communication, as a queue can store messages while the intended receiver is offline. Messages are not limited to a fixed structure and may represent events, data, service requests, or responses. The interacting processes are responsible for interpreting the messages correctly according to the intended behavior of the distributed application. Application fields for message-oriented middleware include the integration of heterogeneous enterprise services in the form of an enterprise service bus (ESB) [Cha04]. In the Actor model [HBS73], concurrent computation can be described by formally specifying the behavior of autonomous actors in reaction to asynchronous messages from other actors.

Coordination middleware based on the *Linda tuple space* concept [Gel85] generalizes the message passing approach by removing the necessity to configure explicit channels between distributed participants. Instead, messages (called tuples) are stored in a shared space that can be accessed by any involved process. Blocking queries on specific kind of tuples enable simple data exchange and synchronization among the concurrently running processes. Tuple spaces provide decoupling in reference, time, and space [Fen04]. The collaborating processes do not need to know each other, do not have to be available at the same time, and need not run on the same machine. Processes are able to join and leave dynamically because the current state of a distributed application can be modeled completely via tuples in the space. Thus, tuple spaces provide not only an abstraction of remote communication, but facilitate *data-driven coordination*, where participants are able to react directly to the shared state of the distributed application. Therefore, space-based middleware is well-suited for the collaboration of autonomous components within an open distributed environment. Figure 1.1 depicts this space-based coordination paradigm, where multiple processes interact by exchanging tuples in a common space.

Linda was designed as a general-purpose coordination language that is orthogonal to established programming languages for computation [GC92]. Thus, its goal is to simplify the specification of coordination logic and enable its separation from the computational aspects of an application. In general, such a *separation of concerns* [Dij82] is highly desirable because it splits complex problems into smaller, more manageable ones that can be addressed independently. However, space-based middleware as well as other previously mentioned approaches typically provide coordination primitives that need to be explicitly invoked by the involved processes. Therefore, developers still have to mix business logic and coordination logic when implementing software components. Thus, separation of concerns is not fully supported. Workflow languages like BPEL [OAS07] or Reo [Arb04] provide an alternative approach, where the coordination logic is specified separately from the application code. Coordination is modeled using a specialized abstraction, which can be based on declarative configurations using a domain-specific language (DSL) or graphical notations similar to Petri nets [Pet66]. On the other hand, the application logic is treated as a black box.

In order to be applicable for real-world applications, middleware has to support multiple phases of software development: design, implementation, and execution. It must be possible to design an abstract model of the solution, which depicts coordination within the distributed application and serves as a reference for implementers of the individual software components. Developers must be able to realize the planned coordination logic using a concrete middleware implementation, e.g., by means of an API, a well-defined DSL, or a graphical modeling tool. Finally, a middleware runtime is necessary to efficiently execute the coordination code, which also requires the handling of remote communication and security.

In recent research, the Space-Based Computing (SBC) Group at the Institute of Information Systems Engineering (formerly Institute of Computer Languages) of TU Wien has investigated space-based middleware technologies for complex distributed systems, with a focus on flexibility, scalability, usability, expressiveness, and interoperability among heterogeneous devices. *Space-based computing* [MK11] refers to data-driven coordination models that rest upon the basic Linda tuple space concepts, but incorporate also features of other middleware types, including message queues as well as relational and NoSQL databases. The *XVSM* middleware (*eXtensible Virtual Shared Memory*) [KRJ05, CKS09] provides a reference architecture for the SBC paradigm. It supports the asynchronous exchange of data entries via structured shared space containers that can be queried by local and remote processes using arbitrary coordination mechanisms, including FIFO queues, key-based access, template matching, and SQL-like queries. A simple API with support for timeouts and transactions enables flexible synchronization and coordination among decoupled distributed processes.

In order to further separate coordination and application logic, a service-based framework termed *Peer Model* [Küh12, KCJ+13] was established on top of distributed space containers. In this middleware concept, the eponymous peers represent autonomous components that communicate with each other in an asynchronous fashion, similar to actors in the Actor model. Within each peer, coordination code and application services

are dynamically triggered depending on the available data entries in its containers. This allows developers to specify complex workflows in a declarative way by configuring so-called wirings that invoke services and migrate data among containers when certain conditions are met. The Peer Model aims to support all phases of software development. A graphical notation enables quick high-level modeling of solutions, whereas developers may choose either a language-independent DSL or platform-specific APIs to implement the coordination logic. The *Peer Space* middleware runtime finally executes the distributed software components on each site. To further ease the design of complex distributed applications, reusable coordination patterns [KCS15] for generic problems like replication or consensus can be defined, which comprise configurable and extensible solutions specified with the Peer Model.

Originally, neither XVSM nor the Peer Model supported security. This thesis investigates suitable security models for these advanced middleware systems and shows how the proposed extensions can be applied to design secure distributed applications. In order to naturally integrate coordination and security, the idea of a "secure room" was used as a starting point, which represents a secured version of space containers. As autonomous processes (in XVSM) and triggered services (in the Peer Model) store their application state and coordinate themselves via these secure rooms, a unified data-driven security concept is possible by regulating the access of the involved stakeholders to the distributed space containers.

1.2 Challenges Towards Secure Coordination Middleware

The main challenge targeted in this thesis is the integration of flexible and expressive coordination capabilities provided by space-based middleware concepts like XVSM and the Peer Model with suitable security concepts, which are not fully available in current coordination middleware. The original Linda model does not address security at all, thus external security measures are required that protect access to each space by intercepting communication with the distributed middleware runtimes. In this case, only coarse-grained security policies can be defined because the space is treated as a black box. Some research has been done concerning security features for coordination middleware, but expressiveness is still limited.

On the other hand, existing security approaches for distributed environments are not designed for usage in coordination middleware. Instead of controlling access to known files and services, transient entries that are part of ad-hoc workflows need to be secured. In some cases, it might be sufficient to adapt established security mechanisms for coordination middleware, but this creates an artificial boundary between coordination and security concerns. Coordination logic specifies how a software component plans to interact with others, while security policies specify how others are allowed to interact with the component. Clearly, these issues are connected, thus an integrated secure coordination approach seems beneficial to simplify application development as well as security management. In the following, the individual challenges towards the design and implementation of such a secure coordination middleware are explained:

- **C1 — Fine-grained access control:** In order to cope with the highly dynamic nature of space-based coordination, coarse-grained access control based on entire spaces or static space partitions (e.g., containers) is not sufficient. In collaborative scenarios, also fine-grained permissions must be supported [TAPH05]. Data entries may represent diverse entities like user data, application state, or service requests. However, due to their transient nature, it is not feasible to set permissions for individual entries. Therefore, it must be possible to specify access privileges based on abstract characteristics of the accessed content (e.g., its data type) and relevant context information (e.g., the current time). As users may dynamically join a collaborative application, they must also be identified in a generic way (e.g., based on their affiliations and/or roles). A suitable definition language for such constraints has to be designed.

- **C2 — Openness:** Open distributed environments often involve clients as well as data and service providers that do not fully trust each other. Sometimes, components may even be hosted by third parties, like in cloud-based or mobile agent scenarios, which requires multitenant architectures that implicate additional complexity. Such open systems can be realized via distributed spaces, whereas each organization should manage permissions to the information it stores [BC01]. Decentralized authorization mechanisms enable autonomous access control for each stakeholder and facilitate secure P2P interactions. Additionally, delegated access should be considered, where a resource is accessed on behalf of a third party [BC01]. For instance, a client `X` may want to invoke a service provided by a server component `Y`, which in turn retrieves data from a backend component `Z` on behalf of the client. Depending on the established level of trust among these participants, the administrator of `Z` may, e.g., allow the described interaction, while direct access by `X` is denied.

- **C3 — Flexible policy administration:** In collaborative scenarios, access control models have to be dynamic [TAPH05]. Administrators must be able to adapt access control policies to cope with new workflows and changed security requirements. In addition, access control should also cover permissions for administrator operations [TAPH05], so that responsibilities can be distributed among several individuals. The separation of concerns between business logic and coordination shall be extended to access control, which should be configurable independently from the rest of the application's implementation. Thus, flexibility regarding changing security constraints can be increased [WPJV03]. This decoupling also makes access control transparent for applications, i.e., modified policies do not necessarily require changes in the rest of the application although they might change the behavior (e.g., by hiding information for which the user is not authorized).

- **C4 — Usability:** Managing access privileges is a sensitive issue, as a single misconfiguration can lead to unexpected behavior and serious vulnerabilities. Therefore, designing a highly expressive security model is not sufficient. It must also avoid

unnecessary complexity, follow comprehensible principles, and provide easy-to-use management operations [TAPH05]. A usable approach should limit the number of concepts an administrator has to learn and ease coordination between security experts and application developers. Therefore, already available coordination concepts (e.g., query mechanisms) shall be reused for modeling security constraints as far as possible. To mitigate the consequences of nonetheless occurring administration errors, layered security strategies shall be considered that provide *defense in depth* [Cab14]. Thus, multiple checks have to be passed in order to gain access.

- **C5 — Middleware runtime integration:** For many collaborative tasks, performance and scalability are critical issues. Therefore, the aforementioned access control features must be enforced by reliable yet efficient middleware runtimes. For some types of devices (e.g., embedded nodes), the full security model may be too complex. Thus, multiple security profiles should be supported, from basic protection to full expressiveness.

- **C6 — Pattern discovery:** It is important to detect solutions for common problems that occur when designing secure distributed applications with the defined security model. This knowledge can be captured in so-called *secure coordination patterns*, which support developers and administrators by suggesting best practice solutions. A suitable structure for a comprehensive pattern catalog must be found, which allows for the composition of basic patterns towards more complex ones.

1.3 Aim of the Work

The main goal of this thesis is to create a suitable security concept for data-driven middleware frameworks that follow the SBC paradigm. The previously defined research challenges can be subsumed in the following research questions:

- What is a suitable abstraction for the specification of access control policies for data-driven coordination middleware?
- How can this coordination middleware be secured in an effective and scalable way?
- How can this secured middleware enable secure collaboration in distributed applications?

These three questions can be mapped to the objectives of this thesis, which address the challenges described in Section 1.2:

- **O1 — A suitable access control model for managing security in collaborative scenarios using SBC middleware:** This access control model must enable stakeholders that participate in an open distributed environment to effectively authorize access to their resources in a fine-grained and flexible way. Solutions to several challenges have to be integrated into a unified concept. The authorization

mechanism must include an expressive language to capture which resources are shared, who is allowed to access them, and under which circumstances access is permitted (C1). Each stakeholder must be able to autonomously specify permissions for direct or indirect access to own resources (C2). These permissions may later be changed dynamically by authorized administrators without requiring changes to the application code (C3). All these concepts have to be combined to form expressive yet usable security features (C4) that fit into the overall middleware paradigm.

- **O2 — Integration of the security concept into heterogeneous SBC middleware architectures:** In order to allow for the development of secure distributed applications, the designed access control model has to be enforced by a secure middleware implementation. Existing SBC middleware architectures like XVSM and the Peer Space have to be extended to handle authentication and authorization. The tradeoff between expressiveness and performance has to be considered in order to design a scalable security architecture (C5). Depending on the capabilities of the targeted platforms and the requirements of intended use cases, different security profiles can be defined. A full-scale enterprise version should support all features, whereas a lightweight embedded version may only require a specific subset.

- **O3 — Investigation of secure coordination patterns:** The specified access control model and its corresponding secure middleware architectures are evaluated by using them to develop complex distributed applications in a secure way. From this experience, secure coordination patterns can be extracted (C6), which provide proven solutions for securing common forms of interaction. These patterns combine established coordination mechanisms with best practices for modeling access control policies, thus ensuring reliable and efficient collaboration, as well as protection against attackers.

The main contribution of this thesis is therefore a security concept for SBC middleware, its realization by means of concrete prototype implementations and the evaluation of its feasibility in practical scenarios. The long-term vision is to provide an alternative, flexible paradigm for designing secure collaborative applications, which gives users detailed control over coordination logic as well as access control and eases development via usable management features and reusable building blocks for secure coordination provided by patterns.

The focus of this work lies on access control, policy management, and delegation concepts for open environments. While other aspects of security like authentication, identity management, and encrypted communication are considered in the overall security concept, this thesis does not claim any significant contribution in these areas. Instead, existing technologies are applied for implementing the prototypes.

1.4 Methodological Approach

In order to identify suitable security mechanisms for XVSM and the Peer Model, an extensive literature study has been performed that targets in particular security concepts for distributed systems and existing secure coordination middleware. The concrete security requirements for SBC middleware were retrieved based on several realistic use cases that were examined as part of ongoing research at the SBC Group, including distributed device management, secure home network automation, and reliable routing in wireless sensor networks.

The security concepts for XVSM and the Peer Model are based on the identified requirements and inspired by existing security models. As the Peer Model is an evolution of the space container paradigm introduced by XVSM, an iterative approach is chosen for the respective security models. First, access control extensions for XVSM are designed according to a novel security model termed *Secure Space*. This concept then also serves as a basis for the more sophisticated *Secure Peer Space*. The secure middleware prototypes are realized as extensions to existing implementations of the respective frameworks.

Based on the targeted use cases and previous research on coordination middleware, recurring coordination patterns and their associated security constraints are analyzed. A selection of secure coordination patterns is defined, for which concrete solutions are modeled and implemented using the introduced secure coordination middleware. Evaluation is performed by means of case studies, where the feasibility of the secure coordination middleware is analyzed with regard to the design of suitable solutions for selected scenarios. In addition, benchmarks are used to evaluate the middleware implementations.

The conducted research was embedded in recent research projects of the SBC Group. During the *Secure Space* project, which addressed the secure management of distributed firewalls, the main concepts for the described security model were devised. In the *LOPONODE Middleware* project, which targets reliable forwarding mechanisms along rail lines using wireless communication nodes, these concepts were refined for the Peer Model and adapted to resource-constrained embedded devices. Additional use cases and coordination patterns were obtained in several smaller projects, including industrial feasibility studies and student projects.

Parts of the work described in this thesis have already been published as articles in international conference proceedings and journals. The most relevant publications are summarized in the following:

- In [CK12], the access control model and security architecture for the Secure Space are introduced.
- This security concept is refined in [CDJK12], which also describes the design and implementation of the secure middleware runtime as well as initial benchmarking results.
- In [CDJ+13], the security model is extended towards a service-oriented architecture on top of XVSM. The introduced Secure Service Space architecture acts as the

connecting piece between the Secure Space and the Secure Peer Space. An example shows its feasibility by means of a use case from the Secure Space project.

- The Secure Peer Space is finally described in [CJK15], which also provides initial examples for secure coordination patterns.

Further publications related to the thesis address the underlying coordination models [KCJ+13, KCJN14, KCH14, CKSW17] and their applications [CKBP14, CHKS14, KCBŠ19], as well as the principles of the used pattern concept [KCS15, KC18]. In addition, the author of this dissertation has co-supervised several diploma theses that are directly or indirectly related to the described work [Win11, Bin13, Bit15, Ham15, Let18, Cej19].

1.5 Thesis Structure

This thesis is structured as follows: Chapter 2 gives an overview on related work for security models and secure middleware. Chapter 3 addresses the applied methods and introduces relevant use cases, while Chapter 4 lists overall design goals. Chapter 5 presents the access control model and the security architecture of the Secure Space as well as its implementation, whereas Chapter 6 shows how the developed concepts can be extended for the design and implementation of the Secure Peer Space. Chapter 7 defines several examples for secure coordination patterns, while Chapter 8 demonstrates the feasibility of the approach for designing practical applications. Chapter 9 evaluates the results of this thesis, whereas Chapter 10 concludes the thesis and gives an outlook to future work. In the appendix, additional syntax and algorithm definitions can be found.

CHAPTER 2

Related Work

This chapter covers related work that is relevant for the thesis' objectives, which can be divided into two major topics: security models and coordination middleware. The following sections define important terms, introduce relevant concepts, and describe concrete technologies. At the end of each section, these technologies are compared with regard to their suitability for secure SBC middleware.

Section 2.1 gives an overview on general security issues, with a focus on access control models and distributed security infrastructures for open environments. Section 2.2 describes how space-based coordination middleware and related approaches deal with security. Finally, the influence of these technologies on the research addressed in this thesis is summarized in Section 2.3.

2.1 Distributed System Security

Security in distributed systems targets three main principles: *confidentiality*, *integrity*, and *availability* [Har10]. Confidentiality guarantees that information is not disclosed to unauthorized users. Integrity prevents illicit modification of messages and data, while availability ensures that access to services, data, and other resources is not interrupted. One major aspect of security is *access control*, which determines how users may access resources and interact with each other. This targets confidentiality by restricting the users who can retrieve certain information, and integrity, as only authorized users may modify (i.e., create, delete, or update) specific data. Additionally, availability measures can be supported by disabling certain attack vectors that require access to internal resources. Access control can be divided into multiple phases [Har10]:

- **Identification**: A user, which can be a system or a real person, has to be linked to a specific *identity* (e.g., an account name). Such a uniquely identifiable user is called a *principal*.

- **Authentication**: When accessing a system, the claimed identity has to be verified based on additional information provided by the principal in the form of *credentials* (e.g., a password or a digital signature).
- **Authorization**: Once a principal is authenticated, the system has to check if it is permitted to perform the requested access operation. The access is authorized if it is compliant with the current *authorization policy* of the system, which is defined by a responsible *administrator*.
- **Accountability**: The principal's activities within the controlled system have to be tracked (e.g., via logging) in order to guarantee non-repudiation. Any access is unambiguously attributed to a specific identity, which ensures that users can be held accountable for harmful actions if necessary.

Other security aspects related to access control are:

- **End-to-end security:** Communication between two or more endpoints (e.g., a user and an accessed system) has to be protected in order to prevent security violations by intermediary nodes. This includes eavesdropping, man-in-the-middle-attacks, replay attacks, and spoofing. Common countermeasures include cryptographic techniques that establish *secure channels* between the communication partners.
- **Trusted computing base (TCB):** This refers to a protected set of software and hardware that the security mechanisms within the system depend on [LABW92]. An important part of the TCB is the *reference monitor*, which intercepts all access to data and services and enforces authorization. Management of authorization policies, cryptographic keys, and passwords may also be included here.
- **Trust establishment:** In an open distributed environment, *trust* between different users is an important issue, as it affects the level of permissions that may be granted. Trust can be established manually (e.g., during the account creation process) as well as using automated mechanisms that rely on dynamic analysis of the detected behavior or on estimations by other trusted sources. Trust is especially important if the authentication process is outsourced to external providers, as the system must rely on the validity of the authenticated user information. Multiple organizations that trust each other may form a *federation*, where the management of identities and the corresponding authentication infrastructure are shared.
- **Privacy:** Users should be able to control the usage of their sensitive data, even when they have to share it with another system to enable a specific service. A privacy policy can restrict usage of this data, so that it may only be used for the intended purpose and must not be forwarded to third parties. Another issue is that uninvolved parties should not be able to monitor a user's activities.

In order to reach specific security goals, further processes and techniques have to be established. This includes technical measures to protect devices and the internal

network, like firewalls, intrusion detection systems, automatic software updates, and malware protection tools, as well as organizational methods like management procedures to handle security violations and corresponding education of employees. These issues are not handled by the security model developed in this thesis and should be addressed independently from the developed middleware system.

2.1.1 Identity Management and Authentication Mechanisms

Any access involves a *subject* (i.e., a user) accessing an *object* (i.e., a protected resource). Before authorization according to the active policy can be performed, the principal representing the subject has to be authenticated using some form of credentials. Typical methods include authentication by knowledge (e.g., a password), authentication by ownership (e.g., an access card), authentication by characteristic (e.g., fingerprints), or a combination of these factors [Har10].

The identities of principals and associated credentials are managed by an *identity provider*. Administrators may create or disable accounts, reset credentials, and link additional *security attributes* like user roles and assigned department to the identity. Such information may be stored in simple files or databases, but it is often organized in hierarchical directories that can be remotely accessed via protocols like LDAP [Zei06]. In distributed systems, authentication mechanisms can be classified according to the relationship between the identity provider and the relying party that hosts the protected resource [CCG+06]:

- **Direct authentication:** Principals are directly authenticated by the accessed host, which means that the system must include its own internal identity provider that stores and verifies the credentials. Web applications often use login masks or HTTP-based authentication [FHH+99] to retrieve user names and associated passwords, which are compared with the stored account information in order to authenticate a principal.

- **Brokered authentication:** A trusted third party server acts as identity provider, which authenticates principals by providing them with a cryptographically signed token that proves their identity. The accessed host is able to verify the token and retrieve the authenticated identity and its associated security attributes without having to cope with identity management. This architecture enables *single sign-on (SSO)*, which means that a principal can log in at a broker and then subsequently access multiple hosts without having to repeat the credentials. Examples for brokered authentication include Kerberos [NT94], OpenID [Ope07], and SAML [OAS05].

- **Offline authentication:** Principals can obtain a proof of identity from a trusted identity provider in advance, which is then presented to relying parties in subsequent communications. Hosts can verify its validity using cryptographic methods and extract the authentication data. In contrast to the brokered approach, the identity provider is not directly involved in the interaction and may also be offline. This

authentication architecture is usually linked with a *public key infrastructure (PKI)* [AL03], where a certification authority acts as identity provider that issues digitally signed certificates as identity proofs. As certificates are valid for a relatively long time and rely on the secrecy of private keys, additional revocation mechanisms have to be supported when a principal is no longer trusted or the keys have been compromised.

In an open distributed system, multiple stakeholders may participate that rely on different identity providers. Therefore, some form of federation between the participants is required, which shares identification and/or authentication information. Each principal must be assigned a federated identity that is unique within the entire distributed environment and propagated in a format that is comprehensible for each participating device, e.g., by means of using standardized protocols like SAML [OAS05]. Authentication mechanisms may even be dynamically negotiated between the involved parties using authentication frameworks like SASL [MZ06].

Workflows often involve multiple stakeholders that are associated with different levels of trust. The path from a request issuer to the final target may include several machines that are not equally trusted [LABW92]. Indirect access is required, where one principal acts on behalf of another one. This is also known as *delegation*. Authorization may depend on a combination of identities for all involved principals [ABLP93]. As a consequence, the authentication identity (i.e., the one associated with the provided credentials) may differ from the authorization identity, which refers to the subject to act as [MZ06]. In this case, additional mechanisms are necessary to ensure that the verified authentication identity is allowed to use the claimed authorization identity. Otherwise, principals would be able to impersonate other users without permission.

2.1.2 Authorization Policies

The authorization policy of a system determines whether the subject that has previously been validated by the authentication mechanism is allowed to perform the requested action. In order to ensure a secure system, access must be limited. By default, subjects should have no access rights. Any permissions have to be explicitly defined according to the requirements of the user and the level of trust the responsible administrator has in this user. Authorization policies should therefore follow the *principle of least privilege* [SS75], which states that users shall only be able to access resources that are necessary for them to fulfill their tasks. As systems usually contain a wide range of different resources, a fine-grained authorization mechanism is required, which restricts access for each user accordingly.

Many authorization techniques are based on the concept of an access control matrix, which is a table of subjects and resources where the values are the corresponding permissions of the subject on the resource (e.g., read or write). For practical reasons, this access control matrix is usually split up according to its rows or columns. An *access control list (ACL)* contains all permissions that belong to a specific resource, whereas a *capability* comprises access rights of a specific subject. While ACLs are usually stored

together with the corresponding resources and checked for every access, capabilities are often provided directly to the user in the form of a cryptographically secured ticket that grants access to the listed resources. An alternative and more expressive way to represent an authorization policy is via a set of *rules*, which define (possibly complex) conditions that determine whether a specific access is permitted or not. Besides subject, resource, and access type, such rules may also target the content of the accessed resource and the context of the operation (e.g., the current time). An example for rule-based authorization is XACML [OAS13].

Authorization policies for a system or a group of associated systems (called *security domain*) may be managed centrally by one or more administrators, which are often also responsible for identity management. They create user accounts and assign appropriate access rights to these subjects. *Administrative policies* regulate access to certain management functions, including changes to the authorization policy. For instance, such administrator privileges may be passed to resource owners that set access rights for their data or managers that control access within their department. Users may also be able to grant a subset of their own permissions to other subjects, which is called *delegation of rights*. In an open distributed system, policy management is decentralized, as administrators for each participant manage their respective security domains. Autonomous authorization policies enable each participant to have its own trust judgment instead of relying on centralized control [VM12].

In large systems with many users, it is not feasible for administrators to set permissions for each user separately. Therefore, several *access control models* have been developed, which provide suitable abstractions for managing a system's authorization policy. In the following, an overview of practically relevant access control models is given.

Mandatory Access Control (MAC). Subjects and objects are assigned specific security attributes, which are used by a centrally managed policy to decide if access should be granted. It is often applied in the form of multilevel security systems for the military sector, where resources are associated with security labels (e.g., "confidential" or "top secret"), while subjects are provided with corresponding security clearances. Subjects must not read information with a security label that is above their clearance. On the other hand, they are also not allowed to write to objects with a security level below their clearance, thus restricting information flow. In order to enforce the least-privilege principle, additional categories can be associated with the resource, which may refer to a project or department. Users must be associated to a matching category in order to use a resource.

Discretionary Access Control (DAC). The creator of a resource (e.g., a file) is set as the owner, who is also responsible for defining permissions for that resource. This is usually done via an ACL that specifies which users (or groups of users) may access the resource in a certain way. A well-known example is the Unix file system, where possible permissions include read, write, and execute rights for files and directories.

Role-Based Access Control (RBAC). In the RBAC model [SCFY96], roles encapsulate subjects with common permissions according to a set of related tasks that they are expected to perform. Instead of directly setting permissions for individual subjects, administrators define permissions for roles (e.g., via ACLs) and then assign subjects to these roles. RBAC is widely used in enterprise applications, where it enables a flexible policy management that corresponds to the organizational structure, as roles often map to job positions and employee turnover does not require complex policy changes. Two main types of RBAC can be distinguished [INC04, Har10]:

- **Core RBAC:** Each subject can be assigned to one or more roles and it obtains their combined permissions.
- **Hierarchical RBAC:** Roles can be organized in a hierarchical structure, where one role inherits permissions from another one. Additionally, constraints may be supported: *Static separation of duty* is enforced by preventing subjects from being assigned to roles that are incompatible with each other. *Dynamic separation of duty* ensures that only one role can be active for a subject at any time. Thus, organizational restrictions can be modeled in a better way.

Attribute-Based Access Control (ABAC). The ABAC model [YT05] generalizes the RBAC approach. Instead of just assigning roles to a user, several security attributes (e.g., department, age, trust level, etc.) can be defined. Access to specific objects depends on attributes of the involved subject, the resource, and the environment. Administrators may enable access only if specific attributes match, or they may even define complex conditions that involve several of these attributes. RBAC and ABAC concepts can also be combined by means of role attributes, dynamic role assignment based on user attributes, or attribute-based constraints on role permissions [KCW10].

2.1.3 Access Control Technologies for Distributed Systems

In the following, relevant technologies for authentication and authorization in distributed systems are described, whose concepts partially inspired the design of a specialized access control model for SBC middleware. This includes well-established standards (SAML, XACML, X.509, and Kerberos) that are exemplary for the current state of the art, as well as two security frameworks (PERMIS and DSSA) that provide additional features related to the objectives of this thesis.

Security Assertion Markup Language (SAML). SAML [OAS05, OAS08] provides an XML-based data format and corresponding protocols for exchanging authentication and authorization data between an identity provider and a service provider that needs to protect its resources. It was established by OASIS as an open standard and is used in commercial products as well as in academic research. The standardized approach enables collaboration among heterogeneous systems and federated identities across different organizations.

SAML assertions are issued by an identity provider and specify different types of statements about a subject:

- **Authentication statements** confirm that a subject has been authenticated at a specific time using a particular method.
- **Attribute statements** provide additional security attributes that are associated with the subject, which is relevant for RBAC and ABAC.
- **Authorization decision statements** declare that the subject is permitted to perform certain actions on a specific resource, which is identified by a URI. For more expressive authorization policies, a combination with additional technologies like XACML is required.

Assertions are embedded in protocol messages, which define queries by a service provider and corresponding responses by an identity provider. SAML provides an abstract framework, as the concrete authentication mechanism is not fixed and different transport mechanisms (e.g., SOAP over HTTP) are supported via bindings. The technology relies on an established trust relationship between the identity provider and the service provider, which is typically based on a PKI combined with digital signatures and secure communication using TLS [DR08]. An extension [OAS09a] also provides chained delegation by including a sequence of intermediaries that act on behalf of an assertion's subject, thus forcing service providers to distinguish between direct and indirect access.

A major use case for SAML is SSO using a Web browser. The service provider redirects the user agent to the identity provider, which authenticates the user and issues a corresponding assertion. This assertion can either be indirectly propagated to the service provider by the user agent, or the service provider retrieves the assertion directly from the identity provider in order to increase security and privacy.

eXtensible Access Control Markup Language (XACML). XACML [OAS13] represents another open standard by OASIS, which targets fine-grained authorization according to the ABAC model. As a specialization, also RBAC is supported [OAS14b]. The standard specifies an XML-based policy language and a corresponding authorization architecture. By providing a common language for authorization policies, access control within an organization can be expressed in a unified and vendor-independent way.

The authorization framework includes the following major elements:

- The **Policy Enforcement Point (PEP)** acts as the reference monitor, which intercepts access requests and grants access when permitted by the framework. In order to obtain an authorization decision, it generates a request that includes relevant attributes related to the subject, the accessed resource, the intended action, and the environmental context.
- The **Context Handler** controls the processing of this request, which is transformed into an XACML request context. According to the resulting XACML response context, the PEP is subsequently notified about the authorization decision.

- The **Policy Decision Point (PDP)** evaluates the request context and computes an authorization decision according to the active authorization policy. For this, the PDP may request the resource content and/or additional attributes from the Context Handler. The decision is then returned to the Context Handler as an XACML response context.

- The **Policy Information Point (PIP)** acts as a source of additional attribute values regarding the subject, the resource, and the environment, which is queried by the Context Handler on behalf of the PDP.

- The **Policy Administration Point (PAP)** enables the management of the authorization policy and makes it available to the PDP.

The policy language is based on declarative rules, which contain a unique ID, a target, an effect, and an optional condition, as well as additional obligation and advice expressions. The target specifies basic matching functions on the provided attributes in order to decide whether the rule applies to the currently evaluated request. The effect determines the outcome of the rule and can be either "Permit" or "Deny". The condition refines the target via more advanced matching functions. Obligations specify additional directives that must be fulfilled by the PEP when granting access (e.g., logging), whereas advices include supplemental information that may be relevant for the PEP.

Rules are stored in Policy elements, which themselves may be structured via nested PolicySet elements. If several rules apply for a specific request, the outcome depends on the defined combination algorithms for rules and policies, respectively. For example, when using "Permit-overrides" for combining rules, the authorization decision resolves to "Permit" if at least one rule with effect "Permit" applies. In contrast, "Deny-overrides" sets the authorization decision to "Deny" when one or more "Deny" rules are applicable. If no applicable rules are found, the authorization decision is "NotApplicable", while a result of "Indeterminate" indicates errors during the evaluation process. In both cases, the PEP should not grant access to the resource. An extension profile for administration and delegation [OAS14a] also allows administrative policies, which specify subjects that may issue additional policies for specific situations. During evaluation, a policy is only considered when the authority of the policy issuer can be traced back to a trusted policy.

The XACML policy language provides high expressiveness, as it supports variable definitions and a wide range of predefined functions (including arithmetic functions, numeric comparisons, logical operators, string matching, and set functions). However, this leads to complex and verbose policies [HFS07], which are difficult to manage without adequate tool support. As XACML is targeted at individual organizations, it does also not address the requirements of "virtual enterprises" with autonomously collaborating subjects [MCSB08], which is required for an open distributed environment.

X.509. The ITU-T X.509 standard [ITU12] is a prevalent PKI technology that is used in many modern Internet protocols like TLS. Like all PKI approaches, it is based on public key cryptography, which relies on asymmetric encryption techniques where each

user holds a secret private key and publishes an associated public key. The private key can be used to create digital signatures by encrypting a hash value of the respective message. The authenticity of the message can then be verified by decrypting its hash value using the public key of the sender. The purpose of the PKI is to establish trust among users that do not know each other. This is achieved via digital certificates that act as proof of ownership of a specific public key. Each certificate includes information about the owner that is confirmed by a trusted third party via its own digital signature.

X.509 assumes that trusted organizations act as certificate authorities (CA), which issue and manage digital certificates. During the registration process, the identity of the requestor and its associated attributes have to be verified, as the CA vouches for their validity by signing the certificate. As a means of authentication, this certificate can then be presented to any communication partner that also trusts this CA. Additionally, a secure communication channel can be established using the public key included in the certificate.

In a distributed system, multiple CAs may be organized in a hierarchical structure, where parent CAs issue certificates for their respective children. Users can verify an identity by checking each certificate along the chain to the root. They only have to trust these root CAs, which provide self-signed certificates that have to be distributed securely in advance. Trust relationship between different CAs can be expressed by means of cross-certificates.

Although certificates are associated with an expiration date, they are usually valid for a relatively long time. If a user is no longer trusted, its certificate has to be revoked. The CA adds such a certificate to its certificate revocation list (CRL), which has to be queried regularly by relying parties in order to prevent access by untrusted subjects. As constant communication with the CA contradicts the concept of offline authentication and would possibly overload the infrastructure, this is one of the main weaknesses of a PKI. For instance, most Web browsers disable CRL checks, so it is possible to set up a secure communication channel with a host that uses a revoked certificate [Har10]. Another problem is the requirement to trust all root CAs. This is a reasonable assumption for a PKI within a single organization, but on a global scale CAs are usually commercial entities that might be subject to intrusions and fraud. A single compromised root CA can compromise the entire system. Thus, it is important to accept only CAs that are trustworthy.

Besides providing a PKI, X.509 also targets privilege management by means of attribute certificates, where trusted attribute authorities (AA) assign security attributes or capabilities to specific users. In combination with a matching public key certificate, relying parties may authorize access to their resources based on this information.

Kerberos. Kerberos [NT94, NYHR05] is a network authentication protocol that follows a brokered approach and relies on symmetric cryptography. A central key distribution center (KDC) holds separate secret keys for each user and service. Using cryptographically secured tickets, it enables mutual authentication and secure communication between users and services.

When a user logs in at the local client, a request is sent to an authentication service at the KDC, which returns a user-specific ticket-granting ticket (TGT) that is valid for a specific amount of time (e.g., several hours). When the user needs to access a specific service, the client sends a request that includes the TGT to a ticket-granting service (TGS) at the KDC. The TGS returns a second ticket that enables the client to authenticate at the targeted service. This ticket may contain authorization data that restricts access to the specified resource. Principals may also allow services to act on their behalf by requesting a special proxy ticket from the KDC.

Kerberos is supported by various operating systems (including Windows and Linux) and is therefore widely used for authentication within heterogeneous company networks. As the KDC constitutes a single point of failure, high reliability and scalability have to be ensured. Kerberos also facilitates SSO in open environments by means of cross-realm authentication, whereas involved realms are listed in tickets to enable individual trust judgments by services.

PERMIS. PERMIS [CO03, CZO+08] is an open source project for a decentralized authorization framework based on the RBAC and ABAC paradigms. It features a modular architecture with an exchangeable authentication mechanism. Identity providers act as attribute authorities that assert attributes about users via X.509 certificates. For each domain, issuing and delegation policies control which attributes can be issued to which subjects and in which form further delegation is permitted. This enables chained delegation, where principals can dynamically delegate some of their attributes to a specific set of subordinate subjects. As the delegation policy is included in the certificate, it can also be validated at the target site. The authorization policy contains two different types of rules that use an XML-based policy language. Trust-related rules define which identity providers are trusted to issue specific attributes, while privilege-related rules specify the actual permissions of the validated subjects. Conditions on context information like current time and access history are supported.

The credential validation service is responsible for checking the local trust-related rules as well as the delegation policies of involved attribute authorities. During the validation, the delegators are recursively examined until a trusted attribute authority is found. The validated subject attributes are then passed to the PDP, which evaluates the privilege-related rules.

Distributed System Security Architecture (DSSA). This security specification for open distributed systems was introduced by Digital Equipment Corporation in 1989 [GGKL89]. It assumes heterogeneous systems, decentralized control, and an absence of global trust. Therefore, each system has its own security policy. The architecture is based on secure channels and authentication via PKI, while authorization is realized via ACLs. As ACLs are treated like regular objects, they enable administrative policies where access to other ACLs (or themselves) can be regulated. DSSA supports both MAC and DAC.

In [GM90], an advanced delegation concept for this architecture is described. It is based on the authentication of delegated systems together with cryptographic assurance that the delegation was authorized by the original user. Users may adopt several roles with differing privileges in order to allow for restricted delegation. Delegations may be chained, whereas each delegated system must be included as a permitted delegate in the ACL of the accessed resource. Lampson et al. [LABW92] provide a formal theory for delegation based on the notion of compound principals, which include adopted roles and involved delegates. ACLs include a formal description of such subjects, e.g., "*UserX* **for** *UserY* **for** (*UserZ* **as** *RoleA*)".

Further Technologies. Besides the described approaches, many additional technologies are relevant for access control in open distributed systems. SPKI [EFL+99] offers a lightweight alternative to X.509 that also includes an authorization policy language. A decentralized PKI approach is followed by the web of trust model [Car00], which relies on a P2P signing mechanism where mutually trusting users sign each other's public key. The WS-Security standard [OAS12b] defines an extension to SOAP that integrates Web services with different security technologies. With the addition of related specifications like WS-Federation [OAS09b] and WS-Trust [OAS12a], a federated environment can be supported.

OAuth [Har12] is a standardized framework for delegated authorization in Web-based scenarios that enables resource owners to permit a client application to access a protected resource on a remote server (e.g., the user's social network profile) without exposing their credentials to the application. OpenID [Ope07] facilitates decentralized authentication in a federated environment, where users can log in to several unrelated sites using the same credentials, which are managed by a third-party identity provider. Its successor, OpenID Connect [SBJ+14], was built as an authentication layer on top of OAuth.

Most of the analyzed systems focus on either authentication or authorization, so a combination of multiple technologies is often required. General-purpose security frameworks like JAAS [Ora19], Shibboleth [MCC+04], and OpenAM[1] enable developers to integrate these access control features into their distributed application. Some security frameworks target specific domains, like Spring Security[2], which provides expressive and customizable access control for Web applications. Restrictions target requested URLs or invoked server methods (possibly depending on parameters or return values).

2.1.4 Evaluation of Security Technologies

The described security methods and technologies are commonly used for access control in large-scale distributed systems, including Web applications, enterprise networks, and cloud environments. In the following, the previously selected approaches are analyzed with regard to relevant challenges for designing secure SBC middleware as outlined in Section 1.2.

[1] `https://www.openidentityplatform.org/openam`, accessed: 2020-04-09
[2] `https://spring.io/projects/spring-security`, accessed: 2020-04-09

Fine-Grained Access Control. It must be possible to formulate complex authorization policies that consider content, context, and subject information. XACML and PERMIS provide expressive rule-based authorization with support for RBAC and ABAC, but only XACML considers content attributes in its policy language. Other systems rely on simple ACLs or capabilities, while context awareness is mostly limited to constrained validity periods. Authentication frameworks like SAML, X.509, and Kerberos provide mechanisms to distribute authorization data, but they do not specify how authorization decisions are obtained. Therefore, they cannot provide fine-grained access control by themselves.

Openness. There is a need for decentralized architectures where users are able to trust each other across organization boundaries. SAML, X.509, Kerberos, PERMIS, and DSSA support multiple identity providers and corresponding trust mechanisms. Delegation on behalf of one or more other principals can be handled via compound subjects, which are fully supported by DSSA. PERMIS facilitates the validation of delegation chains via its trust-related rules. For some other systems, extensions addressing chained delegation have been suggested. This includes SAML conditions for delegation restrictions [OAS09a] and proxying information for X.509 [FHT10].

Flexible Policy Administration. Administrative policies regulate who can manage permissions, thus enabling delegation of rights. DSSA addresses this issue via ACLs on ACLs. XACML provides a mechanism to delegate the ability to define certain policies to specific subjects, but the deletion of policies is not covered. Chained X.509 attribute certificates enable AAs to delegate part of their privileges to other AAs, although this approach is not recommended due to its complexity [FHT10]. In a similar way, restricted dynamic delegation of roles and other attributes is supported by PERMIS, but management of the authorization policies themselves is not addressed.

Usability. Access control models with support for expressive authorization policies should still provide a comprehensible abstraction for application developers and administrators. XACML is very verbose with a lot of predefined features and thus difficult to understand. PERMIS uses a more comprehensible syntax, but still suffers from its complex security architecture with many different components and policies, while DSSA offers a relatively simple model for access control and delegation.

Evaluation Result

As can be derived by this brief evaluation, none of the analyzed systems fulfills all requirements for secure SBC middleware. XACML is not designed for cross-enterprise scenarios and its policies are too complex, whereas DSSA's ACLs are not suited for transient coordination data. The PERMIS approach seems most promising as a starting point, but it still does not support all desired features. The exchangeable authentication mechanism allows adaptation to different scenarios. SAML, Kerberos, and X.509 are

all relevant authentication mechanisms for coordination in open distributed systems. However, PKI solutions have to address the revocation problem in dynamic environments with many users and changing trust assumptions. Proprietary security frameworks combine advantages of multiple authentication and authorization technologies, but are usually difficult to use due to their large amount of features and configuration options. In addition, they mainly target shared files and Web services, but do not consider authorization of coordination data. Therefore, the following section will examine access control mechanisms for space-based middleware and other approaches related to XVSM and the Peer Model.

2.2 Secure Coordination

Coordination and access control are tightly related issues. While coordination makes an interaction fruitful, access control is meant to control interaction in order to make it harmless [COZ00]. Therefore, coordination models for distributed systems should also incorporate suitable access control mechanisms.

In the following sections, different approaches for secure coordination are analyzed. Section 2.2.1 examines coordination middleware based on the Linda tuple space model, which is the foundation for the SBC middleware targeted in this thesis. Section 2.2.2 targets workflow models and service frameworks that are related to the concepts of the Peer Model. Alternative middleware approaches that are relevant to the objectives of this thesis are addressed in Section 2.2.3. Section 2.2.4 includes an extensive comparison of the described systems and also examines reusability by means of secure coordination patterns.

2.2.1 Space-Based Coordination Models and Security

The Linda tuple space model [Gel85] is based on the notion of tuples that are shared among different processes via a common space. A tuple is a sequence of typed fields that may contain arbitrary data. Relevant operations on the space include writing tuples (`out`) and retrieving them in a consuming (`in`) or non-consuming way (`rd`) using template matching. The query operations non-deterministically return a tuple from the space that matches a given template. A tuple matches if type and value of each field are equal to the corresponding field in the template, whereas wildcards are possible that match any tuple field. If no matching tuple is available, the operation optionally blocks until such a tuple is written to the space, which enables simple synchronization among processes, although complex selection criteria like FIFO order or relational operators besides equality checks are not supported.

Several space-based middleware concepts have extended the original Linda semantics. JavaSpaces [FAH99] supports transactions, operation timeouts, and asynchronous notifications. Additional extensions include bulk operations for the selection of multiple tuples [CLZ00, JR06], advanced query capabilities [JR06, Gig19b], and programmable reactions

to events on the space [PMR99, CLZ00, COZ00, JR06]. XVSM, which is described in detail in Section 5.1, incorporates many of these features.

Access Control for Tuple Spaces

The original Linda model does not support security. However, in open distributed environments that include mutually mistrusting entities, a suitable access control model is necessary to preserve confidentiality and integrity of data. Otherwise, malicious processes could eavesdrop on any interaction via blocking queries with wildcard templates, or they could illegitimately trigger certain actions by writing corresponding tuples to the space. Additionally, it would be possible to disturb coordination by deleting tuples or filling up the space with useless data. The authorization mechanism must consider the properties of space-based coordination, where transient tuples are created dynamically by decoupled processes, while providing proper granularity of permissions in order to enforce the principle of least privilege. According to Minsky et al. [MMU00], "segregating communication into multiple tuple spaces increases safety only insofar as it eliminates sharing". As flexible sharing among diverse agents is the main purpose of using a tuple space, the definition of fixed partitions with separate permissions is therefore not recommended. Instead, tuple spaces should support content-based access control to match the nature of the content-based query mechanism.

A major challenge is the openness of the tuple space paradigm, as participants of an ad-hoc collaboration are often not known at design time [FLZ06], but they still need a way to communicate securely via the shared space. For many use cases, a single space is not sufficient. Therefore, access control also has to cope with decentralized spaces that are managed dynamically by local authorities [COZ00].

In literature, several access control models for space-based middleware have been suggested, ranging from simple ACLs and capabilities on the level of fixed space partitions or tuples to complex rule-based policies that consider tuple content and context. In the following, relevant access control mechanisms for concrete systems are described.

Secure Object Space (SecOS). SecOS [BOV99, VBO03] provides access control by protecting tuple fields with locks. Instead of using an ordered sequence, each field is associated with a unique secret label that acts as a key. Any process that wants to retrieve the corresponding information needs to know the respective key, which can be symmetric or asymmetric, and include it in the template together with the associated value (or a wildcard). Tuples may match templates that specify only a subset of their fields, but in this case, the other fields remain invisible to the accessing process. In order to prevent processes from removing arbitrary tuples using an empty template, whole tuples can be protected with a separate lock. By assigning the same key to a group of tuples, a protected partition of the space can be created. The keys, which can themselves be securely distributed within tuples, may be viewed as capabilities that are sent to authorized participants in order to access specific tuples and tuple fields.

SecSpaces. SecSpaces [BGLZ03, GLZ06] follows a similar approach based on the knowledge of secret keys. Instead of labels, special control fields must be matched without using wildcards. Producing and consuming tuples for a specific space partition requires the knowledge of a corresponding partition key. To distinguish permissions for producers and consumers, an asymmetric key is also included in each entry and template, which matches for corresponding public/private key pairs. For example, using the public key of the intended receiver, tuples can be written that are only accessible for this specific user. On the other hand, writing a tuple with the own private key demonstrates its authenticity. Permissions for `rd` and `in` operations can be distinguished by using different keys for reading and removing tuples. WSSecSpaces [LZ04] offers this middleware via a Web service interface, thus enabling the secure coordination of distributed services.

Lindacap. Lindacap [UWJ07] introduces multicapabilities to Linda, which are capabilities for dynamic groups of tuples. They contain a unique tag, a template, and a set of permitted operations. For each space operation, a corresponding multicapability must be presented, whereas the involved tuple (for `out`) or template (for `rd` and `in`) has to match the template in the multicapability. Each tag represents a separate space partition, as an entry is only visible when the tag associated with the access operation is the same as the one used for writing the entry.

The middleware supports multiple tuple spaces that are distributed among several hosts. Processes may create new spaces beside the default one, for which a conventional tuple space capability is returned. Before writing entries to a space, also a corresponding multicapability has to be acquired from the middleware runtime. Any access requires both the capability of the target tuple space and a multicapability for the specific action. These capabilities can be distributed to other processes in order to grant them access, possibly in a limited form with a reduced set of operations or a restricted template. For this reason, a default multicapability enables processes to share capabilities via spaces.

KLAIM. KLAIM [NFP98, NFP99] is a language for programming mobile agents that is based on Linda and process algebra. It supports multiple tuple spaces and operators for managing processes across distributed nodes. Access control is enforced via a capability-based type system, which statically verifies that the intended operations of a process are compatible with the permissions granted by a net coordinator, who administrates the distributed application. Processes, which can themselves be stored within tuples, may only execute if the authorization policy grants them the permission for the corresponding operation type on the target node. It is also possible to specify own policies for dynamically generated nodes. However, KLAIM is not suited for open systems, as all participating sites have to be known at design time [NFP99].

An advanced version has been introduced with μKLAIM [GP03]. It simplifies KLAIM semantics and enables the dynamic modification of authorization policies, which are enforced using a combination of static and dynamic checks. Capabilities may contain patterns, which restrict the allowed tuples or templates within a space operation via template matching [GP04].

Secure Lime. Secure Lime [HR03] extends the Lime [PMR99] model, which adapts Linda to a mobile environment using a federated tuple space. Mobile agents can move among hosts and create new spaces, which are virtually merged with other spaces with the same name on connected hosts. Reactions may be defined dynamically to specify actions that are triggered by the occurrence of tuples that match a specific template.

In the secure version, access control is based on passwords at the level of spaces and tuples. Distributed spaces are only merged if the name and the password used for creating them are equal. Thus, agents can only write and retrieve tuples in remote spaces if they know the associated password. Additionally, this password is used by the middleware runtime to encrypt communication between distributed spaces. Individual tuples are protected via separate passwords for `rd` and `in` that are stored as special tuple fields. The query mechanism only selects tuples that match the password specified for the given operation.

TuCSoN. The TuCSoN [COZ99, COZ00] approach is based on distributed tuple spaces called tuple centres, which facilitate decentralized coordination and security policies. Their behavior can be enriched using reactions that are defined in a logic-based coordination language and stored as meta-level specification tuples. Their triggering condition may depend on the current operation (e.g., type, involved tuple/template, triggering agent) and the presence or absence of specific tuples. Nodes, which may host several named tuple centres, are organized in a tree-like topology, where hierarchic gateways control the visibility of child nodes within their domain.

Access control can be bootstrapped using the reaction layer, which enables the definition of ACLs that regulate agent access on tuple centres and specific tuple types. For this purpose, agents are regarded as composite subjects, which can be expressed via X.509 certificates that identify the agent as well as its enclosing multi-agent system (i.e., the associated application). Different principals for the developer of an agent, the instantiating user, and its current execution environment are considered. Authorization is also enforced for reactions depending on their owner, i.e., the agent that has written the corresponding specification tuple to the space. In order to create nested protection domains, gateways may enforce authentication and authorization on behalf of their child nodes by combining their respective local authorization policies.

In [ORV05], an extended access control model is described that realizes hierarchical RBAC. Instead of simple ACLs, each role is associated with a policy consisting of Prolog-like rules, which enables complex conditions based on the current state of the interaction, template matching on operation arguments, and additional context information provided by built-in predicates. Authorized administrators of an organization may dynamically change role assignments and policies, which is also bootstrapped via a meta tuple centre [OR03].

Law-Governed Architectures. Law-Governed Linda (LGL) [ML95] extends Linda semantics with rule-based access control. A law-governed system consists of a tuple space, a set of processes with associated control states, a global law defining allowed

interactions via Prolog rules, and an associated enforcement mechanism that intercepts space access. Rules may be triggered by write and retrieval operations with specific arguments (invocation events) or by the selection of specific tuples as result of a retrieval operation (selection events). Similar to TuCSoN, events may depend on the content of the involved templates or tuples and on the context (i.e., the control state). As an effect, a rule may allow an operation, modify its arguments, or block it entirely. Additionally, the control state can be updated, which contains arbitrary attributes of the accessing process, including a unique process id and the current time. Using this expressive mechanism, different access control mechanisms can be bootstrapped, including capabilities for allowing interaction with other processes, tuple-based locking, and restricted access to space partitions. Capabilities and keys can be passed as regular tuples and protected with the same mechanisms.

Law-Governed Interaction (LGI) [MMU00] generalizes this approach for an open group of distributed agents that exchange arbitrary messages. LGI is not limited to Linda, but it can be used to protect communication between a tuple space agent and its clients. Rules are triggered by matching specific send and receive events, which contain the sender, the receiver, and the message (i.e., operation type and tuple/template for Linda communication). Access control is enforced by generic controllers running on trusted machines that obtain the global law and the initial control state of their associated agent from a dedicated server.

In order to enable decentralized policies within a coalition of enterprises, an extension of LGI supports policy hierarchies [AM03]. Enterprises are able to define their internal policies independently from each other, but they are constrained by a global coalition policy. Director agents act as certification authorities for their respective domain.

EgoSpaces. EgoSpaces [JR06] is a middleware for context-aware mobile applications that facilitates transiently shared tuple spaces accessed via configurable views. Each agent can specify its own view, which corresponds to a virtual space that contains relevant tuples from connected tuple spaces. View declarations specify a template that matches relevant data tuples, the network distance (according to a configurable metric), and properties of the agents owning the tuples and their corresponding hosts, which are constrained via template matching on their respective profile tuples. EgoSpaces supports transactions and reactions, as well as extended template matching semantics based on configurable constraint functions (e.g., relational operators like "<") instead of simple equality checks. Generic coordination logic (e.g., for data duplication) is encapsulated in reusable components called behaviors.

EgoSpaces manages authorization in a decentralized way, where each agent can dynamically define an access control function that regulates access to its own tuples [JPR05]. This function considers the credentials of the requesting agent and its host, the operation type, the used template as well as the selected tuple, and the owner's profile, which may hold context information. Template matching is used to enable expressive constraints on these properties. The access control mechanism is transparent for other

agents, as they can only retrieve tuples for which they are able to satisfy the access control function of the respective owner.

DepSpace. DepSpace [BACF08] is a dependable coordination service that is formed by several replicated tuple spaces. Byzantine fault tolerance is achieved as space operations can still succeed even if some space servers are compromised. A complex encryption scheme ensures that data stored in tuples can only be recovered by combining information from multiple servers, while still enabling template matching on the encrypted tuples.

The distributed tuple spaces follow the paradigm of Policy-Enforced Augmented Tuple Spaces [BCFL09]. In order to enable consensus algorithms, this Linda variant also supports a conditional atomic swap operation, which outputs a tuple if a specific template is currently not matched. For access control, it uses a fine-grained authorization policy that is composed of declarative rules. These rules evaluate the ID of the invoking process, the operation and its arguments (i.e., tuple/template), as well as the current contents of the space. Additionally, DepSpace also provides a simpler form of access control via ACLs. When writing a tuple, the creator defines two lists that specify which clients may read and remove it, respectively. For each new space, the administrator can specify clients that are able to insert tuples.

For the coordination of Web services, a DepSpace extension has been developed [ABF08], which is based on stateless gateways that act as DepSpace clients.

Lacios. Lacios [ZBH10] is a Linda-like language for modeling multi-agent systems that also includes process calculus elements. Instead of template matching, it uses an expressive query mechanism based on the values of named tuple fields, which supports logical and relational operators, as well as dynamically bound variables. An agent's state (including a fixed identifier) is abstracted via a local tuple. Conditions on this tuple can be formulated similarly to regular queries. For write operations, administrators can define authorization rules that constitute conditions on the state of the writing agent and the corresponding written tuple. For retrieval operations, the respective owner may include similar access conditions that specify who can read or remove the written tuple. This approach is suitable for open systems, as the agents can be described symbolically (based on the properties defined in their state) instead of just by their identifier.

SmallSpaces. SmallSpaces [Fon15] is a tuple space implementation with support for transactions. It provides federated authentication, data-centric authorization via ABAC, and encrypted communication. Each tuple includes subject attributes of its creator and an associated confidentiality rule, which provides a Boolean expression that must be matched by the attribute set of any subject that wants to retrieve the tuple. Consequently, each template used in a read or take operation is enriched with the attributes of the accessing subject and an integrity rule that states which tuples may be considered for the operation according to the attributes of their creators. This enables tuple creators to specify their confidentiality constraints, while tuple consumers can define additional integrity requirements to express their trust in the information providers.

Mutual authentication of clients and the tuple space server is achieved via a PKI that supports distributed identity providers. A signed identity statement is valid for a limited amount of time and may also include a special policy rule that restricts the scope of confidentiality rules for the respective client in order to limit the flow of tuples between domains without mutual trust. Tuple transfer may only be achieved indirectly via a trusted PEP component that takes a tuple and inserts it again with a modified confidentiality rule.

The system also considers the problem of covert channels due to transaction locks. Instead of waiting for the corresponding transaction to finish, authorization is checked immediately on a locked tuple. If the access is not permitted, non-blocking retrieval operations fail immediately. Thus, the client cannot deduce that a matching tuple exists.

GigaSpaces XAP. GigaSpaces XAP [Gig19b] is a commercial distributed in-memory data grid that is based on JavaSpaces concepts. Besides template matching, it includes advanced coordination features like search by IDs, FIFO order, and SQL-like queries. Application data and processes may be distributed across several machines, whereas flexible replication mechanisms ensure high availability. Regarding security, the framework supports authentication, encrypted communication, and authorization [Gig19a]. Administration features enable the configuration of user accounts and roles. Authorization is based on granting users or roles access privileges that define the allowed operations, whereas each space can have a separate security configuration. These permissions are further restricted by class filters that determine the accessible entry types. Higher expressiveness can be achieved via custom access control filters that form an access decision based on authenticated user properties and data fields.

Triple Space. The Triple Space middleware [RMD+06, SKN07] integrates tuple spaces with the Semantic Web by storing interlinked semantic information in distributed spaces. It was developed during the TSC and TripCom projects, which were collaborations of several partners, including the SBC Group at TU Wien. Instead of tuples, the Triple Space stores RDF triples (subject, predicate, object) that may reference each other. In contrast to regular tuple spaces, read operations use semantic queries that yield graphs containing matching triples as well as connected data. Additionally, transactions and asynchronous notifications are supported. Queries can be limited to a single space, otherwise they examine all connected spaces where access is permitted. Spaces are structured in a hierarchy, where triples in local or distributed sub-spaces can be accessed via the respective parent space.

The security model [GCC+07, CCM+08, CCK+09] considers three relevant phases: authentication, trust and attribute mapping (TAM), and authorization. Clients (and also middleware kernels themselves) are authenticated using X.509 certificates via TLS channels. SAML assertions may provide additional attributes about the principal. The TAM component filters attributes and maps them to roles. Its configurable policy determines which attribute providers are trusted to issue which attributes and how specific sets of attributes are mapped to corresponding roles. Transitive trust relationships

between attribute providers can be expressed within SAML assertions. Attributes are accepted as long as this trust chain can be traced back to a trusted authority.

Authorization policies are based on a limited subset of XACML, where certain roles are permitted to perform specific operations on the space. Besides access to triples, also the creation of sub-spaces is controlled. For each space, the authorization policy is set by the owner, which is a special role to which principals can be assigned via space-specific TAM rules. To determine an access decision, combination algorithms are applied that merge the policies of the accessed space and its ancestors. In case of distributed sub-spaces, policies need to be synchronized. Both type of policies are stored in a security space that is only accessible via a separate management API. The security components are decoupled from the rest of the runtime using non-semantic meta spaces that form their input and output stages.

xDUCON. The xDUCON framework [RD09] enforces usage control policies via a flexible space-based architecture. On top of regular access control mechanisms, ongoing management of active sessions is possible that deals with dynamic context changes and supports revocation of permissions. PEP and PDP components access a shared tuple space to coordinate their actions and monitor information regarding the context (e.g., location or time) and the fulfillment of possible obligations. This space supports bulk operations, timeouts, atomic updates, and notifications.

Subjects, targets, and policies are all represented by tuples within the space. Policies are matched by the PDP based on the accessing subject type, the targeted resource type, and the used operation. Within their body, arbitrary conditions on available subject and target attributes are possible. Policies can perform space operations to access context tuples or enable dynamic negotiations of permissions. This also allows them to change subject and target attributes before, during, or after an access. The specified policies can be protected with own mechanisms by interpreting them as resources.

Further Systems. Some additional space-based systems with (limited) access control support are worth mentioning. MARS [CLZ00] is a reactive tuple space middleware on top of JavaSpaces that controls access to its tuples via ACLs. In Tagged Sets [OH05], meta data attached to a tuple can specify symmetric or asymmetric keys that are matched with provided credentials via logical formulas. T Spaces [WMLF98] authorizes user or groups to perform certain operations on a space, which includes the permission to create sub-spaces, thus allowing users to manage their own domains. Finally, Yalta [BGSS01] enables the secure management of dynamic coalitions via coalition spaces and a space-based PKI, where access to each coalition is protected by a CA infrastructure that is implemented by means of multiple components interacting via a common space.

2.2.2 Flow-based Coordination Models and Security

Coordination models can be classified into endogenous and exogenous approaches [Arb04]. Space-based middleware supports endogenous coordination, as processes are able to

interact by invoking space operations within their implementations. In contrast, exogenous approaches define the coordination logic outside of the involved components, thus facilitating strict separation of concerns between computation and coordination. A common way to provide exogenous coordination is to treat components as black boxes with fixed interfaces that can be connected via their output and input ports. Thus, both the *control flow* of a program (i.e., the order of tasks and possible concurrency) and its *data flow* (i.e., the movement and manipulation of data across the system) are defined in a comprehensible way. In the following, such approaches are denoted as *flow-based coordination models*. This relates to component-based software engineering (CBSE) [Crn01] and model-driven development (MDD) [Sch06], as application logic is encapsulated into independent and reusable components, while developers connect them using a DSL or a graphical modeling tool in order to create a complex distributed application.

Flow-based coordination can be expressed at different abstraction levels. Petri nets [Pet66] provide a low-level mechanism to model concurrency in distributed systems by means of a graph that connects transition and place nodes. The semantics of the net can be expressed by means of tokens that traverse the net. A transition fires when tokens are available in all of its input places, in which case tokens are written to its output places. Several extensions have been suggested to increase expressiveness, including Colored Petri nets (CPN) [KCJ98], which attach data to tokens and therefore support more complex conditions, and Timed Petri nets [Bow00], which enable timing constraints.

Reo [Arb04] provides a more high-level approach, where component interaction is modeled based on the notion of channels. Several basic channel types with different semantics exist, like synchronous transmission, content-based filtering, or asynchronous communication via FIFO buffers. In general, channels transport data from a source that accepts data to a sink that outputs the corresponding data, but there are also versions with two sources or sinks, which are used for synchronization purposes. Channels can be combined in a circuit to form complex connectors. Thus, Reo can act as expressive "glue code" for connecting components in CBSE.

A common use case for flow-based coordination is the definition of workflows in a service-oriented architecture (SOA). Services (or tasks) encapsulate specific behaviors and provide well-defined interfaces. Complex distributed applications can be created by means of simply connecting suitable tasks in a structured way, thus forming a workflow. Workflows can be expressed using a Petri net variant termed WF-net [Aal98], where tasks correspond to transitions. However, due to the overall complexity of Petri net models, high-level graphical modeling languages like UML [Obj15] or BPMN [Obj11] are better suited for real-world applications. Such modeling languages provide a clear graphical notation for specifying control and data flow, including support for conditional branches and concurrency constraints.

So far, the mentioned flow-based coordination models focus on the modeling aspect of the software development process. They are mainly used to describe protocols and for evaluation purposes (e.g., simulations or model checking), but not for directly implementing and executing complex distributed applications. Therefore, security is not considered by these approaches. A highly relevant coordination language for implementing workflows

is BPEL [OAS07], which provides a standardized XML-based DSL for the orchestration of Web services using a similar abstraction level as BPMN. To execute a defined workflow, a workflow management system (WfMS) invokes services and evaluates the associated coordination logic according to the specified BPEL code. Security and access control are highly relevant in this case, especially when the involved services are managed by different subjects, possibly from distinct organizations. However, the BPEL standard does not support security constraints and authorization policies. Instead, each participant has to configure local access control mechanisms that do not consider the whole distributed workflow.

Space-based Workflows

Flow-based coordination models can be combined with tuple spaces in order to exploit advantages of both approaches. Space-based coordination may be provided as a Web service that offers an API for a secure tuple space [LZ04, ABF08]. This enables decoupled coordination among multiple components that cannot be easily expressed using direct channels.

Furthermore, tuple spaces can also be used to implement a WfMS that enables high decoupling among participants and dynamic adaptations due to the data-driven interaction. In [ROD02], a workflow engine on top of TuCSoN tuple centres is described. Workflows are modeled by means of coordination rules that are stored as reactions. Dynamic modifications are possible by replacing reactions in the space. A mapping to BPEL exists to enable the orchestration of Web services [CDRV06]. In a similar fashion, 3DMA [FMDV07] defines coordination via active objects that operate on a tuple space. Connector objects realize channels to remote services, whereas router objects trigger connectors and local components according to conditions specified as templates on objects in the space. Dynamic changes to the coordination logic and migration of components are supported. A model-driven design approach is followed by means of UML activity diagrams that are transformed into active objects.

Using distributed tuple spaces, decentralized workflow enactment is possible by splitting the coordination logic of the workflow among the participants [WML08]. Due to the omission of a central coordinator, a single point of failure (or a performance bottleneck) is prevented and the communication overhead is reduced. A suitable model-driven design methodology is proposed in [MWL08]. A BPEL-based workflow model (including deployment information) is translated into a CPN-like variant of Petri nets termed Executable Workflow Networks (EWFN), which are designed to run on space-based middleware. The corresponding process segments are deployed to the distributed workflow engines, which realize the defined EWFN behavior by sharing synchronization tokens and process variables via tuple spaces.

Like BPEL, these space-based workflow models do not address access control. The Peer Model, which is described in detail in Section 6.1, also targets the specification of decentralized workflows on top of a space-based middleware architecture. In order to derive workflow-specific requirements on its access control model, related research on secure workflow models will be analyzed in the following.

Access Control for Workflows

Workflows usually involve many different entities that have to be protected against unauthorized access, including workflow engines, participating services, workflow creators, administrators, as well as repositories for data and meta data. Stakeholders from different companies as well as individuals may collaborate in such a scenario. Thus, inter-organizational workflows require autonomous authorization policies for each component based on individual trust judgments [VM12]. To control dynamic policy changes, meta-level policies may be necessary that regulate permissions for administration of workflows and access control [AT10, TAPH05].

In general, workflow authorization is task-based, i.e., subjects are enabled to perform specific tasks within a workflow [AT10]. Certain dependencies among tasks may be relevant, including *separation of duty* constraints, which ensure that specific tasks are performed by different individuals [BFA99]. Additionally, access may depend on the content of accessed data as well as contextual information like the current time or previous actions. Delegation should also be considered, as clients often invoke services indirectly (e.g., via a workflow engine or intermediary tasks) [LKJZ00]. Following the principles of MDD, also access control constraints can be modeled at design time together with the workflow definition. Using these models, the security configuration can be generated automatically. Such a *model-driven security* approach bridges the gap between coordination and security, as well as the gap between design and implementation [BDL03].

Current access control models for workflows are mostly based on RBAC [LRM11, VM12]. However, classical RBAC does not fulfill all security requirements. In RBAC, separation of duty only ensures that conflicting roles are not activated simultaneously, but it does not prevent a user from performing conflicting tasks in the same workflow. Additionally, an *active access control* model is more suitable for workflows, where permissions are dynamically activated as tasks progress [TAPH05]. In the following, relevant examples for workflow-based access control models are described.

Workflow Authorization Model (WAM). WAM [AH96] is an early workflow-specific access control model that aims at synchronizing the authorization flow with the workflow. Therefore, subjects only gain access to objects that are related to the current task. For each task, one or more authorization templates are defined, which include the authorized subject, an object type, and the permitted operation on the object. An authorization for a concrete object instance of this type is automatically created whenever the task is started within the workflow and revoked as soon as the task finishes. Temporal constraints on task start and end times further restrict access permissions. The model has been formalized using Colored and Timed Petri nets.

In [AH00], WAM is enhanced with RBAC concepts and a flexible mechanism for separation (and binding) of duty. Roles replace subjects in authorization templates and additional constraints can be specified using logical expressions on the history of previous authorizations. Exclusive constraints ensure that no conflicting authorization exists (e.g., the subject must not be involved in previous tasks of the same workflow instance),

while assertive constraints specify that a prior authorization has to be matched (e.g., the subject has to be the same as in a previous task).

SecureFlow [HA99] is a Web-based WfMS that implements WAM concepts. For enhanced usability, constraints can be specified via a GUI by configuring predefined constraint templates that have been implemented using SQL. Administrative policies are supported to protect access to system components, including workflow and policy definition.

Task–Role-Based Access Control (T–RBAC). T–RBAC [OP03] targets enterprise environments by extending hierarchical RBAC with the notion of tasks. Instead of linking roles directly to permissions, roles are assigned to tasks that are authorized to perform certain access operations on information objects (e.g., files or database tables). Thus, users are associated to access rights indirectly via their occupied roles and tasks assigned to these roles.

The model supports active access control for workflow tasks as well as passive access control for stand-alone tasks (e.g., monitoring) that are always accessible for users with corresponding roles. Concerning active access control, authorizations are only valid while a corresponding task instance (with limited execution time) is running. Simple constraints regarding separation of duty can be specified. In contrast to classical RBAC, T–RBAC uses partial inheritance in the role hierarchy, as tasks can be classified as inheritable or non-inheritable. The latter type cannot be performed by parent roles, although they obtain reading permissions on the associated resources for auditing purposes.

Constraint Analysis and Enforcement Module (CAEM). In [BFA99], Bertino et al. describe a constraint analysis and enforcement module that forms the core of a role-based workflow authorization mechanism. It supports the specification of static and dynamic constraints and includes algorithms to check their consistency according to the modeled workflow. A static analysis component checks for conflicts during the modeling stage, while a planning component identifies potential users and ensures that the dynamic constraints are satisfiable. In the run-time phase, these constraints are enforced during user assignment.

When specifying a workflow, each task is associated with one or more roles. Additionally, a set of authorization constraints is specified in the form of logic clauses, which enables complex restrictions on which users are authorized for processing a specific task instance, e.g., to enforce separation of duty. Supported predicates are related to specification (e.g., role/task assignment), planning (e.g., excluding certain users from a task), and execution (e.g., the actual assignment of a task to a user). Additionally, comparison and aggregate predicates (e.g., sum or count) are possible.

An extension to this model [AW05] supports dynamic role activation and conditional delegation. Users may only activate a role depending on specific external attributes (like location) or during certain time intervals. Delegator predicates specify that users want to delegate their rights to process tasks to another user or role, whereas delegatee predicates declare the willingness of accepting such tasks. The validity of these delegations may

be restricted via conditions on the current time, the assigned workload, or specific task attributes.

WIDE. The WIDE WfMS [GPS99] is based on extended database technology with support for advanced transaction concepts and reactive behavior via active rules. It facilitates sequential, parallel, and conditional execution of (possibly nested) tasks. In its access control model [CCF01], task authorizations are assigned to specific roles or organizational levels, which form respective hierarchies. Like in T–RBAC, tasks can hold permissions for specific information objects [CF99].

Dynamic constraints are realized via event-condition-action rules that allow history-dependent, time-dependent, and instance-specific authorization. These rules are triggered by certain state changes (e.g., task executions) or timers and use a logic-based language to specify predicates that may depend on properties of the event and its associated workflow instance. When this condition evaluates to true, permissions for a task (or a specific task instance) may be granted or revoked dynamically in the action part. The derivation of inherited permissions and the actual authorization of tasks can also be bootstrapped via active rules, which are implemented on top of a database using triggers.

To facilitate reuse of active rules in different scenarios, Casati et al. [CCF+00] present a pattern-based development approach. Each pattern specification contains a rule template with parametrized and optional parts, as well as a structured description that defines a name, its intent, usage guidelines, related patterns, a classification, and keywords. Patterns are stored in a catalog that can be used by developers to instantiate concrete rules or design derived patterns via specialization. Examples for authorization patterns are separation and binding of duty [CF99].

AW-RBAC. AW-RBAC [LRM11] provides an adaptive RBAC model for workflows that considers authorizations for security administration and dynamic workflow modifications. The same access control model is applied for task execution, administrative changes like assigning roles and permissions, adaptations of the control or data flow, and replacement of the invoked services.

Permissions specify a category (e.g., Administration), a corresponding object type (e.g., Role), and the authorized operation (e.g., add). These access rights can be refined via constraints on property values of the accessed objects as well as the current time. For instance, only children of a specific role may be affected, or modifications may only be allowed for an individual workflow. Access to a specific workflow instance can be further restricted via a corresponding session that is linked to a separate permission.

SALSA. SALSA [KPF01] is an inter-organizational WfMS that supports fine-grained access control in a decentralized way. It is based on the METEOR workflow model [KFS+99], which itself provides a multilevel security model that controls the information flow among distributed tasks according to security classifications of users and data.

In METEOR/SALSA, a task is connected to other tasks by means of input and output transitions that transport objects of a specific type. Triggering of the task

may depend on guard conditions on the input transitions as well as the selected join mode (all transitions must be activated or only one of them). For a better abstraction of complex workflows, tasks may be nested. Instead of using a centralized workflow engine, a distributed enactment is pursued as each task contains a portion of the workflow specification regarding its interactions with other tasks. Therefore, a SALSA workflow can be split into several autonomous workflows that are managed by different organizations.

Similar to previous models, access control is based on tasks and roles, whereas assignments of authorized roles to a task also contain the corresponding domain for each role (i.e., an organization). When accessing a task, user authentication is achieved via X.509 certificates. For each task, a data access policy defines which data fields within specific objects may be accessed by the task using a certain access mode. Dynamic constraints like separation of duty are realized via a monitor server that logs the execution history and supports necessary run-time queries by tasks. For supporting such constraints across organizational boundaries, multiple monitor servers may be synchronized.

RBAC-WS-BPEL. RBAC-WS-BPEL [BCP06] is an authorization extension for BPEL that combines hierarchical RBAC with dynamic authorization constraints. Permissions to invoke an activity are granted to specific roles, whereas the assignment of users to roles may be determined dynamically based on conditions involving authenticated user attributes, including the employer [PBC08]. For separation and binding of duty, additional authorization constraints can be expressed based on relations of the current user (or its role) with the assigned subject of an earlier activity in the workflow.

Roles and permissions are expressed in an XACML variant, while the constraints are specified in an XML-based constraint language. References to these documents can be added to BPEL specifications and enforced by a PEP that intercepts requests to the WfMS. Besides checking the authorization policy and constraints, the associated PDP also verifies that the requested task assignment does not prevent the workflow from finishing according to the known set of subjects.

SECTET. SECTET [HBAN06] is a framework for secure inter-organizational workflows based on Web services. It facilitates model-driven security, as coordination logic and security constraints are described using a UML-based modeling language that can be transformed into corresponding runtime artifacts. The global workflow logic is split into distributed local workflows that are specified separately by each partner in BPEL. An interface view describes the contract between the participants by means of models for data types (documents), service interfaces, role hierarchies (for each domain), and access control.

Document exchange between partners can be annotated with security requirements regarding integrity, confidentiality, and non-repudiation. Access is restricted using an RBAC approach with expressive constraints defined in the predicative language SECTET-PL, whereas roles are automatically assigned based on the attributes of the accessing subject [AHB06b]. Constraints specify under which conditions a specific role can access a service. They may depend on service parameters, caller attributes, and associated data

from the document model, as well as context information like the current time. Based on the constraint type, inheritance within the role hierarchy may be restricted. These access constraints are transformed into XACML policies and enforced by a security gateway that also enables secure communication with remote workflow engines.

Based on a meta model approach, the framework can be extended with additional functionality. A rights delegation model [AHBU06] describes SECTET-PL constraints for roles that are permitted to delegate access to a specific service to another role. This model can subsequently be transformed into corresponding XACML delegation policies. In a similar way, constrained administrative policies [AHB06a] can be defined to regulate permissions on the interface view, including the ability to change delegation policies.

Secure Business Process Management System (Mülle et al.). Mülle et al. [MSB11b, MSB11a] describe security extensions for a business process management system that cover the entire lifecycle from the modeling to the execution phase. Their security language is embedded into BPMN by means of annotations that can be attached to various BPMN elements (e.g., activities).

Each annotation refers to a constraint type and includes relevant parameters. User and role assignments define permissions to execute associated activities, while mechanism assignments define how tasks should be allocated to users at run time. Constraints for separation and binding of duty are supported as well as delegation of access rights and credentials. Delegation may refer to tasks, but also to access on local data objects or external data stores. The validity may be restricted to the duration of certain activities. The authentication annotation defines a set of attributes required for a role as well as a corresponding identity provider. Additional constraint types regulate confidentiality and integrity of messages and data, auditing policies, and permissions for ad-hoc process adaptation (including data changes). Furthermore, a set of predefined user involvements are supported, which enables users to adapt the workflow and control access to their personal information at run time. This includes the selection of data access and trust policies.

The framework transforms the modeled security annotations into an XACML-based security policy, an adapted workflow (e.g., to integrate user involvements), and a configuration of the involved security components.

SMEPP. SMEPP [BPG+08] is a secure service-oriented P2P middleware targeted mainly at embedded systems. It comprises a high-level service interaction model that allows the discovery and invocation of services within a group of distributed peers. Peers can create or join a group, publish their services, and invoke available services within a group in a synchronous or asynchronous way. SMEPP primitives can be orchestrated into a complex workflow using a BPEL-like modeling language [BP08].

SMEPP features a simple security mechanism based on two types of credentials: an application key for creating a new peer within a distributed environment, and separate keys for joining each protected group. In [BBB+08], this architecture is realized using Secure Lime. Group and service directories are implemented as federated tuple spaces

that are protected with the application key, while the visibility of the individual tuples in these spaces is restricted based on the associated group key. Service calls are realized by exchanging invocation and reply tuples with correlated caller ID in a group-specific tuple space, which is shared by all group members and protected with the group key.

CHOReVOLUTION. The EU project CHOReVOLUTION[3] targets the automatic synthesis of distributed applications by composing decentralized services via a service choreography [SGP15]. It provides an integrated development and runtime environment with modeling notations based on BPMN and support for adaptation, evolution, and security. The CHOReVOLUTION middleware [GBK+15] is based on the notion of the eVolution Service Bus, which enables interaction between heterogeneous peers using a generic middleware abstraction that supports multiple coordination paradigms including tuple spaces.

The security architecture addresses authentication, authorization, auditing, and secure communication among peers and runtime components. For this, standard technologies like WS-Trust, X.509, and TLS are used. The access control concept is based on XACML and thus follows the ABAC model, although not all features seem to be included in the current framework prototype. A synthesized security filter intercepts any access to a service and enforces authentication and authorization. Therefore, it contacts a federation server that supports different authentication mechanisms and maps credentials of the service consumer to valid credentials for the service provider. This server also acts as the PDP, which may grant access depending on provided attributes of the request, the user, and the environment.

Dynamic administration is achieved via an identity manager that controls the identities of users and services, as well as associated permissions. In addition, administrator privileges can be defined to govern access to specific management operations. Multitenancy is supported as each domain has its own policy managed by a dedicated domain administrator.

Further Systems. Besides the previously described access control models, many relevant systems exist that pursue similar concepts. TBAC [TS98] introduced the task-based authorization concept via a dynamic mechanism that controls the lifecycle of permissions based on the progression of tasks in a workflow. WBAC [RDD08] follows a related approach that supports the dynamic acquisition of access rights for workflow tasks. Hung and Karlapalem [HK03] consider authorization at different levels to control the execution of tasks, the generation of events, and access to associated data. In [KS02], the original RBAC model [SCFY96] is extended with the notion of tasks and associated constraints. In a similar way, BP-XACML [ARDS15] considers task instances and separation/binding of duty in an extension of XACML's RBAC profile. Wu et al. [WSML02] present an alternative access control model for METEOR with support for predicates on security attributes of accessed objects. DW-RBAC [WKB07] extends RBAC

[3] `http://www.chorevolution.eu/`, accessed: 2020-04-09

in the context of workflow models with delegation features and logic-based constraints for separation and binding of duty. Tripathi et al. [TAK03] enrich their RBAC model with dynamic constraints for role admission and activation as well as meta-policies for the administration of activities and policies. Administrative policies and dynamic roles are also supported by the access control model of the adaptive WorkSCo WfMS [DRV03].

OrBAC [ACC08], which targets inter-organizational workflows based on Petri nets, supports context-aware access control policies that are enforced by a distributed WfMS. Koshutanski and Massacci [KM03] present a distributed authorization architecture for Web service orchestration with logic-based policies that are defined by the involved partners and combined at the server. Altunay et al. [ABBD05] feature a decentralized access control model based on collaboration policies that indicate mutual trust among participants, including indirect interaction via chained delegation. Narayanan and Güneş [NG11] describe a multitenant access control mechanism that extends the T–RBAC model with task delegation, spatio-temporal constraints, and administrative privileges. In AC3 [YKM14], another T–RBAC extension with MAC-based restrictions on information flow is applied to the domain of cloud computing.

Similar to the work of Mülle et al., SecureBPMN [BHLR12] integrates expressive RBAC permissions into a model-driven security approach based on BPMN. Hummer et al. [HGS+11] describe a context-aware RBAC model with decentralized policies and a SAML-based SSO mechanism, where an enriched BPEL process is generated to enforce authentication and authorization. TABAC [LZS09] enables the dynamic assignment of permissions to tasks, roles, or users in the context of inter-organizational BPEL processes, while supporting several process collaboration patterns that specify how permissions can be assigned and withdrawn between collaboration partners. EB^{3}SEC [JFG+11] follows a model-driven security approach where RBAC is enriched with stateful rules that realize specific patterns (e.g., for ordering constraints or separation of duty) using a formal graphical notation and a transformation into BPEL-based security processes.

According to several surveys [LR14, VM12, EE14], many more access control approaches have been suggested for flow-based models. As their features are mostly similar to concepts already used in the previously described systems, a more extensive analysis has not been performed in the course of this thesis.

2.2.3 Alternative Technologies for Secure Middleware

Communication, data access, and coordination in distributed systems can also be achieved with middleware architectures based on concepts different from tuple spaces and workflow models. CORBA [Obj04] is an object-oriented middleware specification based on the RPC paradigm that enables cross-platform collaboration by means of interoperable Object Request Broker (ORB) implementations. Developers specify interfaces to their objects in an interface definition language, which enables the automatic generation of code for remote invocation. The CORBA specification itself does not address access control, which is instead realized via an external security service [Obj02].

JMS [HBS+13] specifies a message-oriented middleware for Java that enables decoupled communication between two or more clients. It supports two messaging styles. In

the point-to-point model, a message queue is established between two endpoints that are accessed by message producers and corresponding consumers. In the publish-subscribe model, clients subscribe to a specific topic for which messages are published. In contrast to classical message queues, where each message is consumed by a single client, a message is received by all registered subscribers. To increase expressiveness, JMS supports filtering via SQL-like conditions on message properties. However, security is outside of the specification's scope and has to be implemented by JMS providers.

The Actor model [HBS73] provides a formal representation of concurrent computation based on autonomous actors that communicate via asynchronous messages. Decoupling is achieved via mailboxes for each actor that can be addressed by other actors. In response to a received message, an actor can send new messages to itself or other actors, create new actors, and/or change its behavior for future invocations. ActorSpace [AC93] combines these concepts with a Linda-like mechanism for addressing actors. Instead of directly sending a message to an actor's address, one or more recipients are dynamically selected using pattern matching on visible attributes of registered actors in an actor space. A simple capability mechanism enables the authorization of changes to the visibility of actors and actor spaces. Some additional security extensions have been suggested for the Actor model, e.g., based on ACLs [TV03] or MAC [WV04], but the expressiveness of these access control models remains limited.

Access Control for Alternative Middleware Technologies

In the following, additional middleware systems with relevant access control models are described.

CORBA Security Service (CORBAsec). The CORBA Security Service specification [Obj02] extends CORBA with a generic framework for interoperable security features, including authentication, authorization, auditing, non-repudiation, and message protection. Different security levels and technologies are supported, which have to be implemented by ORB providers according to the specified security architecture.

Access control depends on authenticated security attributes of the invoking principal, which may include roles, groups, security clearance, or even capabilities. For a specific operation on an object, an access decision function enforces the authorization policy by evaluating these attributes with regard to control attributes of the target object (e.g., an ACL) and additional context information (e.g., time). Multiple objects can be grouped into a security domain that is controlled by the same policy. Besides access control at the target side, requests may need to be authorized already at the client side. Administration of domains and policies is realized via administrative objects that are protected by security policies themselves.

When object invocations are chained, an intermediate object may use either its own privileges or the delegated credentials of the client. Optionally, also composite delegation can be supported, which combines security attributes of the client and the intermediate object. In this case, the authorization policy may consider security attributes of both principals or even those of all principals along the delegation chain (including the

initiating client). Restrictions regarding the validity period of a delegation, the selection of delegated security attributes, and permitted targets for delegated requests may apply.

A simple standard authorization policy is specified based on an access matrix that grants specific rights on objects to subjects [Kar98]. For invoking an operation on an object, one or more rights (e.g., get, set, manage) are required, which is configured per object interface. Subjects may be abstracted via an attribute (e.g., role), whereas permissions for direct and delegated access can be distinguished. More fine-grained access control can be achieved via security-aware applications that directly evaluate the provided credentials.

Hermes. Hermes [PB02] is a publish-subscribe middleware where events are distributed using a P2P overlay network consisting of event broker nodes. Subscribers register for specific event types and are subsequently notified about matching events from corresponding publishers, which have previously advertised to the framework that they are producing such events. The security concept [BEP+03] is based on the OASIS RBAC architecture [BMY02], which supports role activation via context-aware rules based on first-order logic. Dynamic revocation of roles during a session is possible by monitoring specified membership conditions. Roles can be parametrized, which facilitates that access rights may also depend on additional security attributes of the subject.

In Hermes, each event type has a dedicated owner who is responsible for defining the corresponding authorization policy. Expressive rules define which event types may be sent or received for a specific role. These rules are evaluated during the advertisement and subscription phases at the responsible event broker that is adjacent to the client. If access is partially granted, this event broker stores restrictions in the form of predicates that may depend on certain event attributes or specific context information. Whenever events are sent or received, the framework enforces access control in a way that is transparent for publishers and subscribers. As an optimization, restricted events may be filtered at nodes that are closer to the publisher. Additional policies regulate the connection of clients to brokers and the management of event types. Specific meta events notify brokers about dynamic policy changes. Within the P2P network, event brokers establish trust using X.509 certificate chains. Brokers that are not fully trusted may be restricted to process only a subset of the defined event types.

Pesonen et al. [PEB06] describe an access control extension for multi-domain environments. It is based on SPKI authorization certificates [EFL+99] that act as signed capabilities. Each domain has an access control manager that is responsible for granting privileges to clients and brokers. A type owner can issue delegation certificates to access control managers of trusted domains, which distribute (possibly restricted) capabilities to clients according to their domain-specific authorization policy. By following the certificate chain for a requested operation, the responsible event broker can verify that the delegated authorization was permitted by a trusted entity.

Secure Content-Based Publish-Subscribe System (Opyrchal et al.). Opyrchal et al. [OPA07] present a secure publish-subscribe middleware for privacy-aware event

distribution in pervasive environments. It enables users to subscribe to events that match a specific query, whereas each event is assigned to an owner that controls a corresponding authorization policy. A policy rule includes the authorizing user, a list of authorized subjects, and access conditions in the form of simple logical expressions that must be fulfilled. Conditions target the allowed actions, attributes of the involved event (including the owner property), and context information in the form of external attributes, which are dynamically computed by installed extensions. With the integration of an alternative policy evaluation engine [BZP05], also roles are supported.

For each event type, different security modes can be defined. Access control may be checked already at subscription time or when receiving events (or both). Optionally, encryption and message authentication can be enforced. Via a special action type, users can be permitted to define rules that delegate a subset of their permissions to others. The middleware has to verify that the authorizing user of a rule has sufficient privileges according to previously specified rules (except for predefined administrator permissions).

P-Hera. P-Hera [CSMB05] provides a scalable and secure content hosting platform based on a hierarchical P2P infrastructure. Clients can locate specific data items by querying super-nodes, which maintain indices of available data, and then retrieve it from the target node. Data owners can induce the replication of their content on a remote node according to specific resource constraints.

Clients, data owners, and resource owners (i.e., node administrators) can dynamically establish trust via fine-grained access control based on XACML. Each resource owner specifies policies for hosting and access control. The hosting policy controls which data owners are allowed to place their content, whereas the access control policy regulates access to the stored data. Clients can also define access control policies to prevent getting content from untrusted resources. Placement constraints are specified by data owners and included in replication requests to restrict the nodes where the data should be placed. For increased scalability, these policies are enforced by the super-nodes, which ensure that search and placement requests only return authorized nodes.

2.2.4 Evaluation of Secure Middleware

As demonstrated in the previous sections, various access control models have been suggested to facilitate secure coordination for distributed systems. In the following, these approaches are evaluated regarding relevant features and certain non-functional properties. To enable a comparison of the described systems, several evaluation criteria have been extracted from the overall challenges defined in Section 1.2. These criteria cover specific key aspects that seem crucial for the development of secure coordination middleware. The focus lies on the expressiveness of the authorization mechanism, which is also the main emphasis of this thesis. Following the systematic middleware comparison, additional characteristics of secure middleware are addressed. Subsequently, the relation of secure coordination middleware and patterns is analyzed. A short summary of the evaluation results concludes this section.

Middleware Comparison

Table 2.1 provides the results of the conducted middleware comparison based on the available documentation for the examined systems. It shows how the different technologies (including known extensions) cope with the defined criteria. Fields marked with "+" indicate full conformity, while "∼" and "−" denotes partial and lacking support, respectively. In the following, a description of the individual criteria and rationales for the corresponding ratings are given.

Flexible Coordination. Middleware features should allow developers to realize complex coordination logic in a reasonably high-level way. This includes comprehensive query capabilities, mechanisms to govern control and data flow, and programmable reactions to events. The complexity of the underlying coordination model affects the design of a suitable access control approach. Simpler coordination models are easier to protect against unauthorized access, but they lack the expressiveness to model flexible interactions.

Regarding space-based architectures, TuCSoN is an example for high expressiveness due to the support for logic-based reactions that can be defined dynamically. EgoSpaces and LIME provide a similar (albeit slightly less expressive) reaction mechanism using template matching, while KLAIM and LACIOS combine tuples spaces with process calculus in order to model concurrency. As a commercial product, GigaSpaces XAP supports SQL-like queries that exceed the expressiveness of template matching and provides a wide range of additional features, like for event processing, distributed task execution, and transactions. Due to the support for semantic queries, transactions, and notifications, also the Triple Space fulfills this criterion. Other systems that follow the classical Linda paradigm more closely only allow limited expressiveness due to the lack of reactivity features and their restricted query mechanism. While reaction-like behavior could be realized in LGL/LGI (via laws) and xDUCON (via policies), this is not considered here as these features are designed mainly as mechanisms for access control, which should not be mixed with coordination logic to ensure separation of concerns.

Examples for expressive workflow models are WIDE and SALSA, which support nested tasks and conditional transitions. Model-based systems that rely on BPEL, BPMN, or UML also fall into this category, as well as SMEPP, which supports a BPEL-like orchestration language. Other systems enable the definition of the control or data flow, but do not focus on expressive dependencies or do not support concurrency.

Limited support for flexible coordination is also provided by the examined content-based publish-subscribe systems, whose subscription mechanisms can be compared to asynchronous queries in simple tuple spaces. CORBA provides extensive coordination features (e.g., for discovery, events, and transactions), but its interaction model relies on the relatively low-level RPC paradigm. P-Hera is not designed as a high-level programming abstraction for coordinating processes, as it simply supports distributed access to specific data items.

		Flexible Coordination	Fine-Grained Permissions	Subject Abstraction	Context Awareness	Decentralization	Administrative Policies	Usability
Space-based	SecOS	∼	∼	−	−	−	−	−
	SecSpaces	∼	∼	−	−	−	−	−
	Lindacap	∼	∼	−	−	∼	−	−
	(μ)KLAIM	+	∼	−	−	∼	∼	−
	Secure Lime	+	∼	−	−	∼	−	−
	TuCSoN	+	+	∼	+	+	+	∼
	LGL/LGI	∼	+	+	+	∼	−	∼
	EgoSpaces	+	+	+	+	+	−	+
	DEPSPACE	∼	+	−	+	−	−	∼
	LACIOS	+	∼	+	∼	−	−	∼
	SmallSpaces	∼	∼	+	−	∼	−	∼
	GigaSpaces XAP	+	∼	∼	∼	+	∼	+
	Triple Space	+	−	+	−	∼	∼	∼
	xDUCON	∼	+	+	+	−	+	∼
Flow-based	WAM	∼	∼	∼	∼	−	∼	∼
	T–RBAC	∼	∼	∼	∼	−	−	+
	CAEM	∼	−	+	∼	−	∼	−
	WIDE	+	∼	∼	∼	−	−	∼
	AW-RBAC	∼	∼	∼	∼	−	+	+
	SALSA	+	∼	∼	∼	+	−	+
	RBAC-WS-BPEL	+	−	+	∼	−	−	∼
	SECTET	+	∼	+	+	∼	+	∼
	Mülle et al.	+	∼	+	∼	−	∼	∼
	SMEPP	+	−	−	−	∼	−	−
	CHOReVOLUTION	+	∼	+	+	∼	∼	−
Other	CORBAsec	∼	∼	∼	∼	+	+	∼
	Hermes	∼	+	+	∼	∼	+	∼
	Opyrchal et al.	∼	∼	∼	∼	−	+	+
	P-Hera	−	∼	+	+	+	−	−

Table 2.1: Comparison of secure middleware systems

Fine-Grained Permissions. Access control should be based on individual data items, instead of granting permissions only at the level of applications or fixed data partitions. Expressive content-based policies that are dynamically matched with the accessed object ensure flexibility, as constraints may apply to a set of dynamically instantiated data items with common properties. This is demonstrated by systems like LGL, which supports Prolog-based rules for invocation and selection events that consider attributes of the corresponding tuple or template. All access control models that fulfill this criterion apply some form of predicates or matching functions on attributes of the objects for which access shall be granted. Some systems provide limited expressiveness, where constraints are mostly restricted to simple equality checks. This includes Lindacap, μKLAIM, and the work of Opyrchal et al. For AW-RBAC, the constraint language is not fully specified. In SECTET and CHOReVOLUTION, fine-grained data restrictions are supported indirectly via constraints on parameters of the authorized tasks, albeit they have to be specified separately for each operation.

Several space-based systems, including SecSpaces and Secure Lime, enable the definition of permissions at the granularity of individual tuples, but they do not consider content-based rules or only support them for write operations (like LACIOS). GigaSpaces XAP and WAM support limited dynamic checks based on the object type, whereas T–RBAC and WIDE link tasks to access permissions on named information objects. In a similar way, SALSA grants access to statically defined object fields, while Mülle et al. consider data adaptations and user-defined data access policies. CORBAsec indirectly regulates access to different object operations by associating them with predefined rights, whereas object instances have to be manually grouped into different domains in order to enable separate policies. P-Hera builds on the high expressiveness of XACML policies to control access to specific data items, but does not explicitly address content-based restrictions. In contrast, only coarse-grained permissions are possible for SMEPP (per group) and the Triple Space (at space level). The remaining flow-based models manage only permissions on tasks without considering involved data objects.

Subject Abstraction. Besides specifying the accessed objects in a fine-grained way, authorization policies should also provide a proper subject abstraction to simplify policy management. Instead of having to assign permissions to each user individually, access should depend on roles and other attributes of the subject. XACML-based systems like CHOReVOLUTION and P-Hera naturally follow this ABAC approach. In a similar way, permissions in SmallSpaces and EgoSpaces depend on subject attributes included in a request, while rules in LGL/LGI, LACIOS, and xDUCON retrieve such information from the respective agent's state.

Access control models based on core or hierarchical RBAC only partly fulfill this criterion, as they only take the role attribute into account and rely on the manual assignment of users to roles. However, RBAC and ABAC concepts can be combined, as demonstrated by several systems where roles are dynamically assigned or constrained based on the subject attributes (e.g., Triple Space and RBAC-WS-BPEL). In CORBAsec,

other attributes beside roles are supported, but they cannot be combined to specify a subject in the standard policy, thus providing a similar abstraction as core RBAC.

Context Awareness. Context-aware policies do not only depend on the involved subject, the accessed object, and the used operation type, but also on contextual information. This includes the current state of a distributed application, the history of previous interactions, the relation of the user with the object (e.g., ownership), additional request parameters, and the environmental context (e.g., time or location).

In space-based middleware, context can be represented via tuples in a natural way. TuCSoN, DepSpace, and xDUCON support context-dependent policies that check for the occurrence of such tuples, whereas special predicates or variables can be used to specify conditions that depend on the subject identity. In the case of LGL/LGI, a separate control state is used to store the context, whereas EgoSpaces manages this information in agent profiles. Lacios supports rules that rely on the state of the invoking agent and its relation to the accessed tuple, but does not consider other forms of context. For GigaSpaces XAP, access control can be extended with custom filters that specify conditions on the relation of subject and object.

Sectet policies may refer to context information that is linked within the UML model, while CHOReVOLUTION and P-Hera inherit the context awareness of XACML. Other flow-based coordination models mainly focus on specific context aspects, like history-based constraints for separation of duty. Context information may also be used for access decisions in CORBAsec, although this is not supported by the standard policy. The examined publish/subscribe systems incorporate context via conditions on externally evaluated attributes. However, the expressiveness of these mechanisms is limited (Opyrchal et al.) or mostly unspecified (Hermes).

Decentralization. This criterion refers to systems that support coordination and access control in a P2P style. Instead of having to rely on centralized servers, each peer should be able to host processes and store data locally, and protect access to these resources in a decentralized way. Such an approach is supported by distributed space-based middleware like TuCSoN, EgoSpaces, and GigaSpaces XAP, which allow the definition of separate authorization policies for each space. In a similar way, P2P interactions in SALSA, CORBAsec, and P-Hera are protected. In the Triple Space, each space defines its own policy, but authorization may depend on remote parent spaces.

Sectet and LGI support some degree of autonomy for distributed hosts regarding access control, but they are restricted by a predefined global policy. In a similar fashion, μKlaim supports dynamic authorization policies for each node that are constrained by a central net coordinator. In SmallSpaces, policy rules set by distributed identity providers prevent cross-domain access, while interactions among multiple tuple spaces are not considered. Secure Lime, Lindacap, and SMEPP use passwords or capabilities to control access to distributed tuple spaces. This approach, however, does not specify the actual permissions. The distribution of the credential information has to be implemented explicitly and its propagation cannot be controlled easily. CHOReVOLUTION supports

separate policies for distributed security domains, but stores them on a central server. Hermes enforces access control via distributed brokers, but policies either are defined globally per event type or rely on the manual propagation of certificates. Other systems do not consider decentralized access control, even though some of them (e.g., LACIOS) support access rules set by individual data owners. However, coordination takes place via a single server or, in the case of DEPSPACE, a single logical tuple space that is replicated among multiple hosts.

Administrative Policies. Access may also be regulated at the meta level via administrator privileges that enable dynamic changes to the authorization policy and other configuration aspects of the middleware. AW-RBAC provides a holistic approach via fine-grained permissions on security administration and workflow modifications that use the same mechanisms as regular access control. Other bootstrapped approaches can be observed in TuCSoN, xDUCON, and CORBAsec by means of policies about access to policy tuples and objects, respectively. An alternative mechanism to define administrator permissions is chained delegation of rights, where authorized subjects are enabled to delegate part of their permissions to other users. Hermes, Opyrchal et al., and SECTET follow this approach, whereas the latter additionally considers explicit administrator access to the meta model.

A less expressive way of controlling delegations and process adaptations is also suggested by Mülle et al. CHOReVOLUTION enables the definition of permitted management operations for different administrator roles, but also lacks expressiveness compared to other approaches. For GigaSpaces XAP and the Triple Space, administrator permissions are defined at an even coarser granularity. The WAM implementation SecureFlow also addresses administrative policies, but does not provide any details on their expressiveness. The extended CAEM model [AW05] facilitates temporary delegations of tasks and roles, but administrator access on authorization constraints or workflow definitions is not controlled. Finally, KLAIM provides limited support via capabilities for creating new nodes that can be associated with their own authorization policy.

Usability. The usability of an access control model is mainly determined by the perceived complexity of its authorization policies. In the context of coordination middleware, also separation of concerns between coordination logic and security configuration is important to enable the flexible specification of permissions. The practical usability of a system also depends on how policies are represented and modified via associated administration interfaces and GUIs, but this aspect could not be evaluated because these tools are not publicly available for most of the analyzed systems.

The criterion is fulfilled for systems that rely on ACLs or policies with relatively simple constraints. Examples for comprehensible policies are AW-RBAC and EgoSpaces, which support basic conditions on specific attributes. Some access control models provide higher expressiveness at the cost of simplicity, which could lead to errors in authorization policies when configured by inexperienced developers. They require some programming effort

to specify permissions, using logic-based rules (TuCSoN, LGL/LGI, DepSpace, WAM, WIDE, Hermes), functional programming (SECTET), or imperative decision routines (xDUCON). Mülle et al. support a manageable amount of features to model access control, while an intuitive concept for how to specify the different constraint types is missing. Similarly, CORBAsec defines relatively simple standard authorization policies, but suffers from its complex specification that leaves many implementation details open [BRV02].

Some systems (like P-Hera and CHOReVOLUTION) rely on verbose XACML policies that may be difficult to understand. The Triple Space and RBAC-WS-BPEL specify relatively simple policies, but still lack comprehensibility due to their usage of XACML. CAEM relies on a complex calculus that makes it unsuitable for end users [BFA99], similar to Klaim. Lacios and SmallSpaces use comprehensible conditions, but fail the separation of concerns aspect by setting the constraints within the write operation, which means that access control has to be managed within the application code. Other approaches (e.g., Lindacap) require the explicit distribution of keys or capabilities, which mixes up security and coordination logic even further.

Further Characteristics of Secure Middleware

The goal of this thesis is to design a secure coordination middleware that supports all mentioned evaluation criteria. However, there are also several other relevant properties that have been discovered in related work.

Nested protection domains, as realized via TuCSoN gateways or sub-spaces in the Triple Space, enable the definition of permissions in a hierarchical way, with general rules at the upper layers and more fine-grained ones near the bottom. Several independent security domains may also be managed at the same host, thus allowing for multitenant scenarios. This can be achieved via separate partitions that are controlled by their respective creators (like in Lindacap), or via restricted administrator privileges that only apply to a specific domain (like in CHOReVOLUTION).

The management of cross-organizational interaction requires federation mechanisms that involve independent identity providers with potentially different levels of trustworthiness. The Triple Space supports the specification of transitive trust relations among distributed identity providers using SAML. In SmallSpaces, the federated identity providers act as trust anchors for their respective communities, similar to the domain-specific access control managers in Hermes. CHOReVOLUTION deals with heterogeneous credentials via a federation server. Several additional systems (e.g., SALSA, SECTET, Mülle et al.) address cross-organizational workflows by supporting multiple identity providers and/or organization-specific domains.

Some systems, like EgoSpaces and TuCSoN, consider compound subjects, but complex delegation chains like in DSSA (e.g., "A **for** B **for** C") are not supported. Permissions depend on the accessing client, but not on the initiator of a workflow or the trustworthiness of intermediate nodes. Only few approaches (e.g., [ABBD05]) even consider trust issues related to indirect access. CORBAsec outlines composite delegation modes that forward security attributes of all involved principals, but still does not specify how to impose conditions on the structure of a delegation chain in an authorization policy.

Task-based access control in several workflow models defines permissions for task executions and — indirectly — also for data access. A related approach is also followed by TuCSoN, which protects the triggering of reactions and controls what kind of tuples can be retrieved or written by them (based on the reaction owner).

As a static network topology with homogeneous nodes can usually not be assumed, support for mobility and interoperability becomes relevant. Several middleware systems (e.g., LIME, EgoSpaces, SMEPP, CHOReVOLUTION) target mobile devices and ad-hoc networks, while CORBA enables cross-platform coordination by means of interoperable middleware implementations. To cope with the limited hardware resources of some embedded devices, lightweight variants of SMEPP [VAC08] and LIME [CMMP06] have been developed for usage in wireless sensor networks (WSN). Providing an expressive access control abstraction for such a downscaled middleware constitutes a difficult challenge.

In space-based middleware, bootstrapping is a common method to simplify the access control model by reusing already established concepts. This includes the application of template matching for authorization (e.g., Lindacap, EgoSpaces) and the usage of tuple spaces to realize the security architecture (e.g., Triple Space, xDUCON).

In order to decouple coordination logic from permissions, a transparent access control approach is followed by several systems, including SecOS, Lindacap, and SmallSpaces. As queries only return data that the subject is authorized to access, coordination logic usually does not have to be adapted when permissions change. On the other hand, users cannot conclude on the existence of protected data based on denied access attempts, as inaccessible entries are simply invisible to them.

Secure Coordination and Pattern Support

The development of complex software can be simplified by reusing established best practice solutions in the form of patterns, which provide structured descriptions for different problem categories that include the respective purpose, suggested solutions, and possible applications. Similar to software design patterns [GHJV95], which target object-oriented software architectures, also coordination patterns [Tol98, HCY99, DWK01] can be defined that focus on the interaction of components in a distributed system. Such patterns provide reusable blue prints for recurring coordination problems [KCS15], thus offering high-level abstractions for developers of distributed applications.

In the context of space-based middleware, coordination patterns have been introduced by Tolksdorf [Tol98], who has demonstrated his approach by means of a few simple patterns targeting mobility in information systems. GigaSpaces XAP provides a more extensive pattern catalog that describes generic best practice solutions based on real-world use cases [Gig17]. Other examined systems do not explicitly support patterns, but offer solutions for recurring coordination problems in the form of bootstrapped components (e.g., behaviors in EgoSpaces) or example descriptions (e.g., an electronic marketplace in KLAIM [NFP98]).

In the context of WfMS, Workflow Patterns [AHKB03] have been established for describing common control flow constructs. For message-oriented middleware, Enterprise

Integration Patterns [HW04] identify different ways of routing messages via queues. They have also been applied for the automatic generation of adapters in CHOReVOLUTION [SGP15]. However, the combination of coordination patterns and security remains an open research issue.

Security patterns [SFH+06] address a wide range of security mechanisms, like session management and secure channels. Uzunov et al. [UFF15] present a more specific pattern catalog that describes sensible features for distributed authorization infrastructures. In [MMD+14], a pattern refinement method is used for extending SECTET with additional security mechanisms. All of these approaches focus on the architectural level, i.e., on how certain security features can be realized within a middleware framework. However, patterns that provide guidance for the definition of suitable authorization policies on an existing middleware platform for specific collaborative scenarios have not been extensively researched yet, even though such secure coordination patterns could ease the development of secure distributed applications.

Several evaluated systems provide some examples for how to use their respective coordination and access control mechanisms to perform collaborative tasks in a secure way (e.g., a secure bidding mechanism for LGI [MMU00] or a partial barrier with DepSpace [BACF08]), but they do not follow a systematic and structured pattern approach. WIDE provides a pattern catalog [CCF+00] that includes a few authorization patterns, but they focus on rather basic constraints (e.g., binding of duty) instead of complex coordination. Other pattern-based approaches targeting authorization in distributed systems [LZS09, JFG+11] have similar limitations. Thus, there is a lack of structured descriptions of secure coordination patterns that specify solutions to generic coordination problems together with recommended authorization policies.

Evaluation Result

In summary, several access control models for expressive coordination models have been identified that fulfill most, but not all, of the defined evaluation criteria. Compared to established general-purpose security technologies, the examined access control models are generally less mature, but in many cases, they still enable more expressive authorizations due to their direct integration with the coordination functionality. Open research topics remain in the areas of delegated access (with support for complex subjects), fine-grained meta policies for the dynamic administration of the system at run time, and secure coordination patterns.

2.3 Related Work Summary

By examining the state of the art in access control mechanisms for distributed systems in general and coordination middleware in particular, several desirable features have been identified that have an impact on the design of a security model for SBC middleware as targeted by this thesis:

- To support complex security requirements, expressive rule-based access control models with fine-grained permissions via content-based and context-aware constraints are beneficial, as demonstrated by systems like XACML, TuCSoN, and LGI.

- Recent approaches are usually based on variants of RBAC or ABAC, which provide good abstractions for controlling access by a specific group of users.

- It cannot be assumed that a distributed application is controlled by a single organization. Therefore, decentralized policies and support for federated identities become important. Trust-based rules, as available for PERMIS and the Triple Space, enable permissions that depend on the established trust in distributed identity providers.

- Due to the nature of loosely coupled ad-hoc collaboration with changing participants and dynamic modifications of the coordination logic, also authorization policies must be configurable at run time. However, such administrative operations also have to be protected by access control. Possible solutions are the controlled delegation of own privileges (as used by Opyrchal et al.) or the explicit definition of fine-grained administrator permissions (like in AW-RBAC).

- Access decisions may not only depend on the actual invoker, but also on the initiator of a request and involved intermediate nodes, as suggested by the chained delegation mechanisms of DSSA and CORBAsec.

- The tradeoff between expressiveness of the access control model and its usability has to be considered. Authorization policies should be easy to comprehend for administrators in order to reduce errors during the specification and subsequent modification of permissions.

- A modular framework approach as followed by PERMIS and CORBA enables a flexible security architecture that reduces dependencies between its components. Different authentication mechanisms like SSO or PKI may be used according to the targeted application scenario.

- Pattern support fosters reuse of generic solutions for specific problems. While WIDE follows a pattern-based development approach and also supports some patterns for authorization constraints, the definition of high-level patterns for secure coordination, which combine coordination patterns with associated authorization policies, is still an open issue.

As the coordination models of XVSM and the Peer Model exceed the expressiveness of classical tuple spaces and workflow models, respectively, the evaluated access control models cannot be utilized directly. Systems with comparable coordination features, like GigaSpaces XAP (for XVSM) and SALSA (for the Peer Model), use relatively simple

access control models that do not fulfill all relevant requirements. Therefore, adapted solutions for the desired features are necessary, which will be presented in this thesis.

The novelty of this approach lies in the integration of expressive data-driven coordination models with customized access control mechanisms and in the description of secure coordination patterns that demonstrate best practices for defining corresponding authorization policies in specific collaborative use cases.

CHAPTER 3

Methodology

In this chapter, the applied methods and techniques for achieving the research objectives from Section 1.3 are described. For the development of suitable access control models for XVSM and the Peer Model, respectively, four phases can be distinguished: *requirements analysis*, *access control model design*, *prototype implementation*, and *evaluation*.

An iterative approach is pursued, where XVSM is targeted first, followed by the more elaborate Peer Model. As the Peer Model builds on XVSM concepts, results from the requirements and design stages of the Secure Space can be reused as a foundation for the Secure Peer Space. The requirements and evaluation phases involve a set of realistic use cases with complex coordination and security constraints, which originate from application scenarios described in Section 3.1. Preliminary evaluation results regarding these use cases have led to several refinements of the original designs and prototypes.

Finally, as a result of the continuous application of these secured middleware variants in practically relevant scenarios, secure coordination patterns are identified in the *pattern specification* phase. In the following, the outlined phases are described in more detail.

Requirements Analysis. The requirements of the targeted access control models are based on several real-world use cases that involve complex interactions in an open distributed environment with different stakeholders. Following a standard software engineering approach, the desired functionalities of these distributed applications are defined in corresponding use case specification documents. Based on these documents and interviews with the respective domain experts, the functional and non-functional requirements of the applications are derived, which includes necessary countermeasures to identified security risks. As secure coordination middleware should provide features that simplify the development of such applications, these use case-specific security constraints constitute requirements on the security model of the middleware. Due to the focus on access control, mainly requirements concerning authentication and authorization are considered for this thesis.

As a limited set of use cases cannot cover all possible scenarios for which secure coordination middleware may be applied, further design goals have been identified via an extensive and systematic literature review, whose results have been described in Chapter 2. Furthermore, existing experience on space-based coordination within the SBC Group is leveraged by analyzing possible security constraints for common coordination patterns. The thereby derived requirement specification forms the starting point for the design phase.

Access Control Model Design. For both XVSM and the Peer Model, suitable access control models have to be designed that integrate the specified requirements into the respective coordination models. The existing middleware specifications are therefore extended with corresponding mechanisms that are adapted from state-of-the-art concepts identified during the literature review. The focus lies on the authorization phase, as the special coordination paradigms of these systems require sophisticated solutions in this area that cannot simply be adopted from existing approaches. Identity management and authentication are not explicitly specified, but the integration of suitable external mechanisms into the corresponding access control models and the overall system architectures are described. Accountability measures are not directly addressed, but the integration of logging mechanisms for failed and successful access attempts appears trivial for the respective solutions.

The authorization policy as well as corresponding API extensions and data types are described using a DSL approach, where the syntax is specified in EBNF-like notation [ISO96]. An informal graphical representation connects authorization rules with the corresponding coordination logic. The middleware architecture and the semantics of the model are partially bootstrapped by means of meta-level components that interact via the respective middleware. The internal behaviors of components are defined in a detailed way, using UML diagrams [Obj15] to describe complex processing logic. A more formal specification, e.g., using operational semantics, is out of scope for this thesis, as it relies on the underlying formalization of XVSM and the Peer Model, which is ongoing research.

The described approach is first applied to XVSM and then to the Peer Model, which provides flow-based coordination on top of XVSM-like containers and therefore can also leverage the XVSM access control model. As a connecting link, the *Secure Service Space* [CDJ+13] is defined, which facilitates a simple workflow model on top of the Secure Space. In order to demonstrate the modularity of the approach, an adapted version of the Secure Peer Space for resource-constrained embedded devices is outlined.

Prototype Implementation. In order to enable the practical evaluation of the designed access control models, corresponding proof-of-concept prototypes are developed. They are based on existing research prototypes developed at the SBC Group, which have been realized for XVSM [Bar10, Dön11] and the middleware architecture of the Peer Model, the Peer Space [Cej19]. As these middleware runtimes were written in Java, also their access control extensions are developed using this object-oriented programming

language, which features high portability, concurrency support, and extensive library functions. As an additional proof-of-concept, a restricted version of the Secure Peer Space has also been developed for the .NET framework in C♯ within the scope of a master thesis [Bit15] co-supervised by the author of this dissertation.

The abstract access control architecture defined in the design stage has to be adapted to the concrete middleware architectures of the respective prototypes. This includes the integration of authentication and authorization components into the middleware runtime, the mapping of the DSL specification to equivalent APIs, and optimizations regarding performance and scalability. Due to performance considerations, the Peer Space implementation uses its own lightweight container implementation instead of XVSM. Therefore, there is also no overlap in the secured middleware implementations, although an alternative prototype version of the Secure Peer Space could be bootstrapped via secure XVSM containers.

Based on the concrete middleware architectures, the access control features are realized by modifying and extending the respective prototype implementations. The functionality is verified by means of extensive unit and integration tests.

Evaluation. In this phase, the feasibility of the designed access control models and their corresponding proof-of-concept implementations is analyzed, mainly with regard to expressiveness, usability, security, and performance.

Expressiveness and usability are evaluated by mapping the features and characteristics of the presented access control to the previously defined requirements. In addition, the systematic middleware comparison from Section 2.2.4 is extended with the secure versions of XVSM and the Peer Space, which relates them with existing technologies for secure coordination. Practical applicability is also analyzed by means of case studies, where specific coordination tasks have to be modeled and/or implemented with the suggested secure middleware systems. These tasks target the examined application scenarios as well as additional problems identified during requirements analysis. External validation is performed via several master thesis projects [Sch13, Bin13, Kan15, Let18] that use the developed middleware extensions to develop complex distributed applications with fine-grained access control. The practical realization of these case studies with reasonable effort regarding the definition of authorization policies indicates the overall feasibility of the suggested approach. A more in-depth analysis would require a series of extensive usability tests with a wide range of participants, which is, however, out of scope of this thesis and subject to future work.

The security of the approach is evaluated by analyzing its compliance with established principles and guidelines from literature. In addition, possible attack scenarios and corresponding countermeasures are discussed. Micro benchmarks on the implemented prototypes measure the overhead of the access control features on basic middleware operations. These benchmarks incorporate authorization policies with varying complexity and different amounts of protected data, thus analyzing the scalability of the solution as well as the tradeoff between expressiveness and performance.

Pattern Specification. The results of the case studies form the knowledge base for the identification of secure coordination patterns, which are derived from the developed solutions. These patterns shall provide generic templates for suitable, middleware-based solutions to common coordination problems together with appropriate authorization policies for all participants.

The structured description format for secure coordination patterns shall be based on related pattern methodologies [GHJV95, CCF+00, SFH+06], while the generic pattern solutions are specified according to a pattern-based approach on top of the Peer Model [KCS15], which supports configurable parameters and pattern composition. XVSM-based patterns are not explicitly addressed but can be largely subsumed by corresponding interactions with the Peer Model.

As a complete pattern catalog would exceed the scope of this thesis, only a selection of relevant patterns is specified, which may act as the foundation for an extensive pattern language. The functionality of these patterns is validated via implementation with the previously developed Java Peer Space middleware. The feasibility of the pattern-based approach is demonstrated by showing how use cases can be realized via the combination of multiple patterns, whereas only a minimal amount of additional coordination logic is required.

3.1 Application Scenarios

In the following, an overview of the examined application scenarios and their corresponding security constraints is given. These scenarios comprise a wide range of use cases where coordination middleware like XVSM and the Peer Model can significantly ease the modeling of complex and dynamic interactions. The access control models developed in this thesis shall enable developers and administrators to control these interactions also from a security point of view.

3.1.1 Management of Distributed Firewalls

The goal of the Secure Space[1] research project was to provide middleware support for the secure management of several distributed firewall devices using a common application termed *Security Management Center* (SMC). The SMC acts as a proxy that supports administrators with the management of firewall configurations, the supervision of the network, and other relevant features. Firewalls are a critical part of an organization's infrastructure and an erroneous configuration of a single device may compromise the security of the whole network. Compared to the direct management of individual firewalls by local administrators (e.g., for each department), a centralized management approach ensures consistency among all configurations and enables a fast reaction to observed threats. The functionality of the SMC can be classified into three main categories: *configuration*, *deployment*, and *monitoring*.

[1] `http://www.complang.tuwien.ac.at/eva/projects/secure-space.html`, accessed: 2020-04-09

Configuration features assist administrators with the generation of concrete firewall settings based on generic policies. While it is also possible to modify and store configurations for individual firewalls via the SMC, the main advantage is the capability to define templates that cover a whole class of firewalls. Such templates provide abstract configurations with generic parameters that are dynamically evaluated by the SMC for each associated firewall. These parameters refer to variables on the SMC, properties of the respective device, or complex rules that depend on several values. Additionally, dependencies between multiple firewalls (e.g., for the creation of a VPN tunnel) can be expressed in a high-level way via managed objects, which affect certain settings for each involved firewall. The SMC also checks the consistency of each modification before allowing the generation of the concrete firewall configurations.

In the deployment phase, the computed configurations are simultaneously pushed to the managed firewalls, which accordingly update their internal data models. To ensure a consistent state of the distributed environment, transactional guarantees should apply. If not all updates are successful, the whole operation should be rolled back.

The monitoring features provide administrators with detailed information about their network infrastructure. This comprises configurable statistics and events pushed by the firewalls (e.g., error messages or intrusion alerts), as well as remote access to the current states of individual devices, including their active configurations and dynamic information like throughput or CPU load.

The involved SMC modules should interact in a highly decoupled way in order to allow for flexible deployment options and extensibility. Depending on the involved stakeholders, the SMC approach can be applied to different settings:

- In the *single organization* setting, the SMC is responsible for firewalls of a specific organization. One or more administrators have full access to the SMC, which in return controls the distributed firewalls within the company network.

- In the *service provider* setting, the firewall administration is outsourced to an external service provider that manages firewalls for multiple companies. This may be suitable for small enterprises that do not want to employ their own security experts. SMC administrators configure firewall policies separately for each organization, but synergies may emerge due to the shared security administration, e.g., malware outbreaks may be detected earlier and critical updates could be pushed to all connected firewalls at once.

- The *distributed administration* setting provides the most complex scenario. The overall firewall policies are again managed by a central provider, but also the administrators of the individual organizations have limited access to the SMC. These local administrators may monitor their own firewalls and change certain settings as long as they are not in conflict with the overall policies. For instance, a state-wide education authority may want to control the firewall infrastructure of all schools within their reach to enforce standard policies depending on the school type and ensure up-to-date protection mechanisms, but local administrators

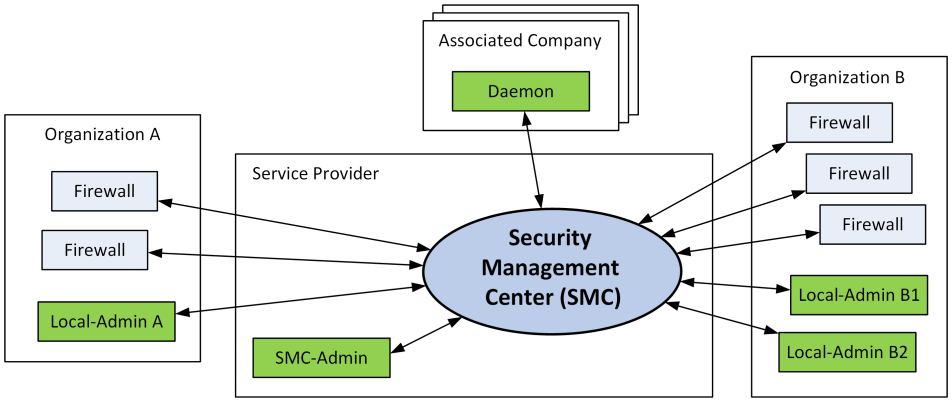

Figure 3.1: Example SMC scenario with distributed administration setting

should have the liberty to also define school-specific rules (e.g., concerning blocked traffic). Additionally, third-party providers may have limited access to certain data available at the SMC. This could be used, e.g., for a distributed intrusion detection mechanism that collects anonymized monitoring data from multiple SMCs [Win11].

The last scenario, which is outlined in Figure 3.1, requires fine-grained access control to regulate the permissions of all involved stakeholders. Therefore, it acts as the initial motivation for the creation of the Secure Space and subsequently also the Secure Peer Space.

Security Constraints

A solution for this scenario must support access control for the administrative interfaces of the SMC and the firewalls. The internal firewall logic is not addressed, as these devices use separate mechanisms (i.e., firewall rules) to regulate access to their respective network segments. The following requirements on the access control model can be derived:

- Subjects shall be abstracted via their organization and role. For each organization, different administrator roles may exist with varying privileges, e.g., only senior administrators may be allowed to modify templates.
- Permissions may be set for specific SMC functions or ensuing operations (e.g., read, update, or delete) on the underlying data. To enforce data restrictions, SMC functions should use the identity of their invoker.
- Data access may be restricted to certain sub-trees of the hierarchical data model (e.g., only specific configuration parts) or depend on specific data values (e.g., only anonymized statistics).
- Local administrators should only be able to access functions and data related to their own firewalls.

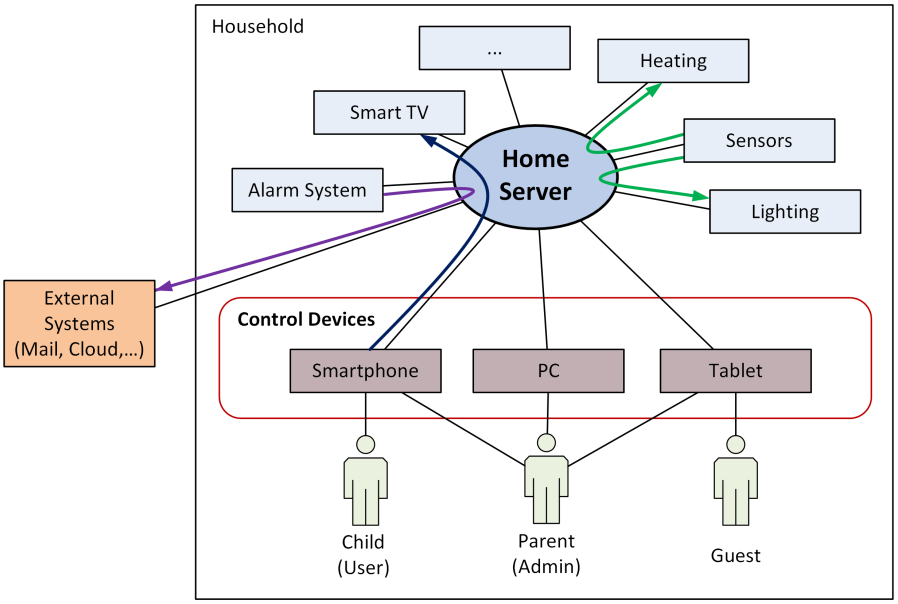

Figure 3.2: Smart home scenario with configured workflows

- Direct and indirect access must be distinguished. To ensure consistency, administrators should not be able to change firewall settings directly, but only via the SMC.
- Mutual trust establishment between the SMC and its managed firewalls has to be achieved at run time. Only registered firewalls may interact with the SMC and each firewall may only accept commands from its dedicated SMC.
- Firewalls must not be able to forge messages to the SMC so that they appear to come from different devices (e.g., to cause reconfigurations at the network of a competitor managed by the same SMC).
- Dynamic changes to the authorization policy of the SMC should be possible.

3.1.2 Smart Home Management

Within the scope of a feasibility study, the *infotope*[2] technology for secure management of devices within a smart home environment has been examined. It is based on a private cloud concept with a home server that controls access to managed devices, like TV sets, smartphones, alarm systems, sensors, and light switches, as well as interactions between them. By connecting heterogeneous devices in a unified and secure way, a flexible form of home automation can be achieved.

Figure 3.2 gives an overview on the targeted scenario. Via the home server, which is remotely accessible through control devices like PCs, tablets, or smartphones, authorized

[2] `http://infotope.com/`, accessed: 2020-04-09

users can monitor and manage their devices. Common interactions can be automatized by specifying workflows that involve local devices as well as external targets (e.g., cloud storage or mail notifications). These workflows may be triggered by specific device events or via user input through a control device. For instance, a user may request an update about the status of all managed devices, or a motion sensor may activate the heating system if the presence of any resident is detected. A user-friendly GUI should enable the simple configuration of workflows and access privileges.

As this scenario focuses on secure workflows involving distributed devices and multiple stakeholders, it motivates the usage of the Secure Peer Space for its implementation.

Security Constraints

The envisioned security features include fine-grained access control for network configuration, device access, and workflow invocation. As the functionality of the managed devices cannot be modified directly, all checks have to be performed within the implementation of the infotope home server, which acts as a proxy for interactions within the home network. Several requirements regarding access control have been identified:

- An RBAC approach should be followed, including roles for administrators (e.g., parents), regular users (e.g., children), and guests.
- Administrators should be able to configure users and add devices. Devices may be shared among all users or assigned to a dedicated owner.
- Workflows shall be specifiable by administrators and invokable by authorized users.
- Users shall be able to dynamically specify restrictions for access to their personal devices. This includes control over the devices' participation in workflows, depending on involved users and devices.
- Machine-to-machine traffic shall only be allowed if it is required for a workflow.
- The functionality of the home server shall only be accessible via trusted control devices (e.g., with an installed certificate).
- Some permissions may only be valid within a certain time window.

3.1.3 Wireless Sensor Network for Railway Applications

The LOPONODE project[3] comprised various research efforts concerning the applicability of WSN technology for use cases in the railway domain. The main goal was to replace expensive cabling along railway tracks with wireless communication via a network of smart *low-power nodes* (*LOPONODEs*) that are powered by small solar panels, while preserving relevant QoS properties like high reliability and low latency. Each of these

[3] `http://www.complang.tuwien.ac.at/eva/projects/loponode.html`, accessed: 2020-04-09

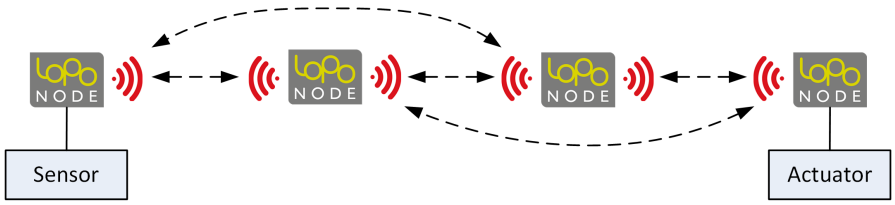

Figure 3.3: LOPONODE forwarding chain with redundant forwarder nodes

embedded devices provides basic radio functionality for point-to-point transmissions and broadcasting, whereas end-to-end routing of information can be flexibly specified at the application layer depending on the requirements of the concrete application. Some LOPONODEs interact directly with their environment via sensors and/or actuators, while others merely act as forwarder nodes.

The motivating use case was an autonomous safety system for railway crossings, where sensors have to transmit train detection events to a controller at the crossing site over a distance of several hundred meters, so that corresponding signals can be activated in time. As the radio range of LOPONODEs is limited, such messages between endpoints have to be relayed via a chain of forwarder nodes, as depicted in Figure 3.3. To ensure a fast, reliable, and efficient connection, suitable communication protocols are necessary that incorporate repeated transmissions and redundant nodes. Similar mechanisms can also be applied to other relevant use cases, like avalanche warnings and train arrival announcements.

The Peer Model is well-suited for modeling such complex communication protocols, thus determining the coordination of LOPONODEs within a specific application scenario. However, the LOPONODEs must also be secured to avoid interference by unauthorized subjects. For closed systems like in the railway crossing use case, a simple cryptography-based approach with a shared key may be sufficient to prevent access by any node that is not part of the current crossing configuration, but other scenarios could involve LOPONODEs owned by different stakeholders that do not fully trust each other. For example, an application that notifies about the upcoming arrival of certain goods at a cargo terminal may not only involve the railway infrastructure company, but also different train operators and logistics companies. Thus, fine-grained access control is also relevant for the embedded version of the Peer Model.

Security Constraints

In order to establish a secure WSN, access control has to be enforced on each distributed LOPONODE, so that only legitimate interactions are possible. For the envisioned use cases, the following constraints can be derived:

- The identity of a LOPONODE can be abstracted based on its owner (i.e., a company) and its role within the application scenario (e.g., forwarder or sensor node).

- Permissions may depend on the message type and on the current node state.
- Secure routing mechanisms have to be considered, as two endpoint LOPONODEs may need to communicate via a chain of potentially untrusted forwarder nodes. A corresponding delegation mechanism has to be designed, so that permissions depend on both the transmitting forwarder node and the originating endpoint.
- Authentication should be based on symmetric cryptography, which is directly supported by the LOPONODE hardware and can thus be executed more efficiently than alternative methods.
- The usage of resource-constrained embedded hardware requires simple access control mechanisms with minimal usage of CPU time, memory, and bandwidth.
- To ensure the fulfillment of memory and timing constraints, all necessary data structures must be statically allocated and all security checks must be processed in a bounded amount of time.

3.1.4 Additional Use Cases

XVSM and the Peer Model have been applied to a wide range of additional scenarios that were examined during the course of research projects with industrial partners as well as small-scale student projects and academic case studies based on well-known coordination problems from literature. This ongoing research mainly targets the domains of collaborative P2P services, cloud computing, and reliable infrastructure for intelligent transportation systems. As security is highly relevant in these scenarios, their constraints have also affected the design of the access control models and patterns presented in this thesis, although the solutions to these problems cannot be described in detail here.

CHAPTER 4

Requirements

According to the proposed methodology, requirements for a suitable access control model have been compiled based on constraints of the examined reference use cases, previous experience with the development of distributed applications using space-based middleware, and problems discussed in related work. As both XVSM and the Peer Model use a data-driven approach based on the concept of space containers and target similar application scenarios, a common set of requirements for their access control models can be specified. However, due to differences in their programming paradigms and middleware architectures, these overall design goals may be implemented in different ways.

The derived requirements for suitable access control extensions are listed in the following, grouped according to the main challenges from Section 1.2. The first five requirements relate to fine-grained access control (C1) and the expressiveness of the model:

- **REQ-1 — Fine granularity:** ACLs at the container level are not sufficient, as entries may represent different kinds of data (e.g., public/private messages, service requests, or internal states) with varying authorization constraints. Simply putting these entries into separate containers with different permissions would create a rather strict dependency between access control and coordination. In order to ensure confidentiality and integrity without restricting coordination logic, entry-specific permissions shall be possible for write and query operations.

- **REQ-2 — Content-based access control:** The investigated middleware technologies are mostly used for coordination via transient entries. Due to their short lifetime, it is not feasible for an administrator to set permissions for each entry individually. Including permissions directly when writing an entry would violate separation of concerns, as application developers would be forced to also handle access control. Therefore, content-based authorization rules with suitable expressiveness for secure data-driven coordination are necessary, where permissions are

specified based on one or more properties of an entry (e.g., its type or a user-defined priority).

- **REQ-3 — Context awareness:** Access control shall consider relevant information regarding the context of a requested access. Expressive conditions on the current application state and the environmental context (e.g., CPU load), which can both be represented via entries in specific containers, need to be possible in authorization rules. Request context data, like attributes of the invoking subject and the current time, shall also be able to influence access decisions. Private entries shall be enabled, which can only be read or removed by their creator (or, e.g., by someone from the same organization), without having to specify separate rules for each individual.
- **REQ-4 — Subject abstraction:** The access control model shall support scenarios with dynamically joining and leaving users that are not known in advance. User identities shall therefore be described in a symbolic way within authorization rules. For instance, at least role and organization attributes must be provided in the SMC use case.
- **REQ-5 — Completeness:** Access control must not be limited to data access operations, but has to cover the complete functionality of the middleware. This includes the invocation of hosted services and the configuration of the coordination logic. A unified mechanism should cover authorization for all access types in a similar way.

The next four requirements target the suitability for open distributed systems (C2):

- **REQ-6 — Decentralized authorization:** In order to support P2P interactions without a central server, access control shall be configured and enforced in a distributed way. This means that each participant must be able to autonomously define its own authorization policy.
- **REQ-7 — Delegation support:** Policies shall distinguish direct from indirect access, where a subject acts on behalf of another one (e.g., an SMC accesses a firewall on behalf of an admin user). Permissions may not only depend on the identity of the actual invoker, but also on other participants that have been involved in the current interaction.
- **REQ-8 — Federation:** In an open distributed system, it usually cannot be assumed that there is only a single identity provider with a fixed authentication mechanism. Thus, the access control model shall support different identity providers that issue subject information in a consistent way. Each stakeholder must be able to decide on its own which forms of authentication shall be trusted.
- **REQ-9 — Multitenancy:** To support collaboration within distributed applications, multiple stakeholders have to access a shared middleware instance. However, mutual trust among them cannot be assumed, so their actions and hosted data

have to be isolated from each other unless explicit permissions are granted. In order to enable organizations to control access to their own data, independent security domains with separate administrators shall be supported.

The following requirements address flexible policy administration (C3):

- **REQ-10 — Dynamic modifications:** The SBC paradigm enables the adaptation of distributed applications with additional or modified components at run time. Also security requirements may change over time due to new user roles or discovered abuse of permissions. Therefore, the access control mechanism shall cope with dynamic authorization policy updates without requiring a restart of the system.

- **REQ-11 — Administrative policies:** Remote management of permissions shall be possible according to configurable policies. Privileged users shall be able to delegate some or all of their administrative rights to other subjects.

- **REQ-12 — Decoupling:** The coordination logic should be mostly decoupled from the active authorization policy. To some extent, it should be possible to change permissions without affecting the coordination code and vice versa. Therefore, transparent access semantics must be applied, where denied entries are hidden while accessible entries that also fulfill the specified query shall be returned instead. The code of the accessing component does not have to be changed if denied entries appear like missing entries, which already have to be dealt with due to the space-based coordination.

Additional design goals target the prevention or mitigation of configuration errors, thus addressing usability (C4):

- **REQ-13 — Comprehensible policy language:** Authorization policies should be easy to understand for application developers and security administrators. The combination of multiple authorization rules shall be possible in a structured way with clear semantics.

- **REQ-14 — Bootstrapping:** Already existing features of the coordination model should be reused for access control in order to provide a natural integration of coordination and security.

- **REQ-15 — Layered security:** Multiple protection layers shall be supported that target the different abstraction levels of space-based interactions, i.e., access to network, services, and data. When operations require permissions on multiple levels, illicit accesses cannot be enabled by a single misconfiguration.

Regarding middleware runtime integration (C5), the following requirements have been identified:

- **REQ-16 — Trustworthy middleware runtime:** The defined access control model must be strictly enforced by a middleware runtime that cannot be bypassed or compromised by external attackers. Unauthorized access must be prevented in any situation, so that users can fully trust the access control mechanism.
- **REQ-17 — Scalability:** As high-speed coordination is often crucial, the access control model must not excessively degrade the performance of the middleware runtime. The tradeoff between complexity of policies and execution time of operations has to be considered. The access control mechanism should scale well with the number of managed entries and provide the option to optimize performance if not all policy features are required.
- **REQ-18 — Modular architecture:** To facilitate extensibility, the security architecture should provide appropriate extension points and consist of decoupled modules that can be exchanged according to the present application scenario. This supports the creation of custom-tailored middleware solutions for specific scenarios.

The final requirement supports the definition of secure coordination patterns (C6):

- **REQ-19 — Reusability:** Most forms of collaboration in a distributed application can be reduced to a few common coordination patterns. To foster reuse of secure and dependable solutions, a straightforward mechanism to incorporate access control into such patterns is needed.

CHAPTER 5

From XVSM to the Secure Space

The space-based middleware XVSM (eXtensible Virtual Shared Memory) [CKS09] combines concepts of Linda tuple spaces with advanced coordination features for the flexible interaction of distributed components via P2P communication. It is based on the notion of structured space containers [KMS08, KMKS09], which constitute shared sub-spaces with separate coordination laws, like FIFO queues, dictionary access, or template matching. These coordination mechanisms can be combined and extended with customized coordination logic in order to realize complex distributed applications, whereas the general advantages of the data-driven Linda model, like high decoupling and a simple API, are retained.

In an open distributed environment, the collaboration among the distributed participants has to be secured. The Secure Space concept extends the XVSM middleware specification with access control features that prevent misuse of middleware functions and protect confidentiality and integrity of managed data. In Section 5.1, an overview of the underlying XVSM middleware is given, which is specified in more detail in [Cra10]. Section 5.2 describes the suggested access control model [CK12, CDJK12], while specifying the syntax and semantics of its authorization policies. Section 5.3 shows how this access control model is integrated into the XVSM middleware architecture [CDJK12] in order to provide mechanisms for managing and enforcing authorization. This architecture can be used to bootstrap a secure distributed workflow model termed Secure Service Space [CDJ+13], which is outlined in Section 5.4. Relevant information about the Java-based implementation of the Secure Space can be found in Section 5.5. Section 5.6 describes related benchmark results, while Section 5.7 analyzes the approach and indicates open issues.

5.1 XVSM Overview

In XVSM, coordination is achieved by means of *entries* that are shared via distributed *containers*. A process may use local containers via its own embedded XVSM *space*

Operation	Parameters	Result
write	*ContainerRef*, *EntriesWithCoData*, *OpTimeout*, *TxID*, *Context*	-
read	*ContainerRef*, *XQuery*, *OpTimeout*, *TxID*, *Context*	*EntryList*
take	*ContainerRef*, *XQuery*, *OpTimeout*, *TxID*, *Context*	*EntryList*
createContainer	*SpaceRef*, *ContainerName*, *ContainerSize*, *CoordConfig*, *TxID*, *Context*	*ContainerRef*
destroyContainer	*ContainerRef*, *TxID*, *Context*	-
lookupContainer	*SpaceRef*, *ContainerName*, *TxID*, *Context*	*ContainerRef*
createTransaction	*SpaceRef*, *TxTimeout*, *Context*	*TxID*
commitTransaction	*SpaceRef*, *TxID*, *Context*	-
rollbackTransaction	*SpaceRef*, *TxID*, *Context*	-
addAspect	*SpaceRef*, *IPoint*, *AspectImpl*, *Context*	*AspectID*
removeAspect	*SpaceRef*, *AspectID*, *Context*	-

Table 5.1: Relevant functions in XVSM Core API

instance and/or access containers on remote spaces. The query capabilities of a container depend on the associated *coordinators*, which provide separate views on the managed entries. For writing and querying entries, the middleware provides Linda-like operations with blocking semantics and support for *transactions*. *Aspects* can be used to enrich the behavior of these operations with customized logic. In the following, these basic concepts and their realization via the XVSM middleware runtime are explained.

5.1.1 XVSM Core API

Processes can interact with local and remote spaces via the *XVSM Core API* (*CAPI*), which is depicted in Table 5.1. It provides operations for container access as well as for the management of containers, transactions, and aspects. The table shows the required parameters and result values with their respective data types (for a detailed declaration see Section A.2.1 in the appendix). Besides the given return values, each of these operations may also fail and indicate an exception to the invoker. Below, these data types and operations are described.

Entries. The basic element of XVSM is an entry, which contains a set of key-value pairs termed *properties*. These properties contain application-specific data used to store information or coordinate with other processes. In contrast to tuple fields in Linda, they are unordered and can be accessed via their distinct key. Properties may hold objects with primitive or complex data types (e.g., strings, integers, device configurations). Additionally, lists and nested properties can be expressed. The syntax of an entry adheres to the following structure:

$$[key_1\colon value_1,\ key_2\colon value_2, \ldots],$$

whereas key_i is an identifier and $value_i$ can either be a simple value, a nested set of properties (with the same syntax as a full entry), or an ordered list of values in the form "$\langle x_1, x_2, \ldots \rangle$". A formal syntax definition is given in the appendix in Section A.1.2.

As an example, the entry "$[x\colon$'abc', $y\colon \langle 24, 42\rangle$, $z\colon [z1\colon [z2\colon \texttt{true}]]]$" shows the different property types. Simple properties are addressed via their unique property name (e.g., $x =$ 'abc'). Specific values within a list can be accessed via an index notation[1] (e.g., $y[1] = 24$), while for nested property access the names are chained (e.g., $z.z1.z2 = \texttt{true}$).

Spaces and Containers. Each space can be addressed via a unique space reference (*SpaceRef*) in the form of a URI. If no space is specified in a CAPI operation, the local space is implicitly selected. A space may host several containers with globally unique container references (*ContainerRef*), which consist of the space reference and an automatically generated ID. Processes may dynamically create and destroy containers on local or remote spaces. An optional container name allows for the dynamic lookup of containers created by different processes.

The main function of containers is to provide coordinated access to its entries. The `write` operation stores a list of entries into a container, while the other access operations support queries on the set of available entries. An optional size parameter limits the number of entries written to a container. Entries can be retrieved in a non-consuming or consuming way via `read` or `take`, respectively. The semantics of these query operations depends on the defined coordinators, which are set separately for each container in its coordinator configuration (*CoordConfig*). A coordinator definition includes the coordinator name and possible configuration parameters.

Coordinators and Selectors. A coordinator provides a specific view of a container according to its inherent coordination law. In order to maintain this view, some coordinators require their own internal data structures, like a queue or map. Therefore, each coordinator provides corresponding *accountant functions* (`register` and `unregister`) that are automatically invoked whenever entries are written to or removed from the associated container. When writing to a container, additional coordination data (*CoData*) for each entry specifies parameters for the registration functions of involved coordinators (if required by the coordination law, e.g., a key for the `key` coordinator). Certain coordinator-specific constraints may prevent a write operation from succeeding, e.g., in case of duplicate keys.

For query support, each coordinator provides a corresponding `select` function, which filters and possibly resorts a list of entries according to the internal coordination law and selection parameters specified by the user via a *selector* object. Table 5.2 provides an overview of available coordinators with their registration and selection parameters. This list can be extended with customized coordinators to increase middleware expressiveness.

[1] In this notation, indices start with 1.

Name	Registration	Selection	Coordination Law
any	-	count	indeterministic selection
fifo	-	count	FIFO queue
lifo	-	count	LIFO queue
vector	index	index, count	selection based on list position
key	key	key	dictionary access with unique keys
label	label(s)	label, count	category-based selection via labels
type	type[2]	type, count	selection based on entry type
linda	-	template, count	Linda-like template matching
query	-	query	SQL-like query on entry properties

Table 5.2: Predefined coordinators with parameters for entry registration and selection

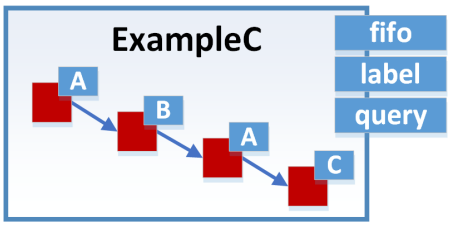

Figure 5.1: Graphical representation of XVSM container with three coordinators

Most selectors feature a `count` parameter, which indicates how many entries are expected in the result. If not enough entries are available, the selection fails. The special value `ALL` indicates that all matching entries shall be returned, which may also correspond to an empty list. For count value `MAX`, the selection is restricted to all accessible entries, i.e., those that are currently not locked by a concurrent transaction. If no specific value is given, the default value of `ALL` is used. Some coordinators (`any`, `fifo`, `lifo`) provide an implicit order that does not depend on other parameters. Others, like `vector`, `key`, `label`, and `type`, attach specific meta data to managed entries by which they can be retrieved. The selector of the `linda` coordinator uses a template that is compared with all entries in the container. Each template field refers to a required entry property, for which either a concrete value or a wildcard that matches any value is given. Thus, entries can be retrieved based on their application-specific data. For more expressive queries, the `query` coordinator should be applied, which enables predicates on arbitrary entry properties. This includes comparisons (e.g., "$price \leq 100$"), set operators, property-based sorting, logical operations, and a flexible entry count.

Figure 5.1 provides a graphical representation of an example container with name "*ExampleC*", where four entries are implicitly ordered in a queue according to the `fifo` coordinator and user-defined labels have been attached by the `label` coordinator. In addition, content-based queries are supported via the `query` coordinator.

[2] An object-oriented XVSM implementation can implicitly provide this information to the coordinator. In the formal model, the type is represented as an additional entry property.

The query operations of the CAPI use an *XVSM query* parameter (*XQuery*), which represents a chain of one or more selectors separated by the pipe symbol ("|"). Each selector comprises the name of its coordinator and the required selection parameters. With regard to the container from Figure 5.1, possible examples are "`fifo`(3)" for selecting the first three entries or "`label`('A', `ALL`)" for returning all entries with label "A". Following the pipes and filters pattern [BMR+96], each selection stage (i.e., a coordinator's `select` function) uses the output of the preceding phase as input, which enables the combination of multiple coordination laws. While the first coordinator operates on the whole container, subsequent coordinators select from the already restricted subset of entries that was returned by the previous stage. If any selection within the chain does not succeed (i.e., not enough matching entries are available), the whole query fails. As an example, the following XVSM query may be used to find new alerts with high priority:

$$\texttt{type}(\textit{Alert}) \mid \texttt{query}(\textit{priority} > 4) \mid \texttt{lifo}(1)$$

First, all entries with type `Alert` are selected (using the default count). These entries are then filtered according to the given query, which matches entries that have a property with name "priority" and an associated value of greater than four. Finally, the newest entry that matches the first two selectors is returned.

Blocking Behavior. To enable flexible synchronization among decoupled processes, container access operations may block if they can currently not be completed successfully. This is mainly useful for the query operations (`read` and `take`) in order to wait for entries from other processes, but also `write` operations may block, e.g., when the container is full. The timeout parameter (*OpTimeout*) specifies how long the operation should be retried before an error is returned to the invoker. This is specified as a time interval (in ms) or using one of the following special values:

- **INFINITE**: No timeout is specified, i.e., the operation blocks until it can be fulfilled.
- **ZERO**: The operation does not block at all, i.e., it is never retried.
- **TRY_ONCE**: The operation is not retried when it fails, but it may wait for resources locked by concurrent transactions.

Transactions. In order to group several CAPI operations into a single atomic action, transaction parameters are supported for container access and management operations. The transaction model relies on pessimistic concurrency control, i.e., entries are locked when accessed within a transaction, thus ensuring atomicity, consistency, and isolation. A common use case are atomic updates of entries, which can be achieved by combining corresponding `take` and `write` operations in a single transaction. Depending on the internal logic of the involved coordinators, locked entries may cause a query operation to block or alternative entries may be selected instead. Indeterministic coordinators like

`any` or `linda` can thus improve concurrency and prevent conflicts by ignoring locked entries as long as matching alternatives are available.

A transaction can be created for a specific space with an optional timeout (*TxTimeout*) after which it is automatically rollbacked. The returned transaction ID (*TxID*) can be used for transactional container operations and to rollback or commit the changes. If no transaction is specified, an implicit one is created that only spans the current operation and is automatically committed afterwards (or rollbacked in case of errors).

Context. Each CAPI operation supports an additional context parameter, which may hold properties injected by the user or the middleware itself in order to provide configuration settings or additional information for specific internal components. This mechanism can also be exploited to inject authentication data into the middleware.

Aspects. The functionality of CAPI operations can be enriched via aspect-oriented programming [EFB01]. In XVSM, aspects constitute user-defined behavior that is injected before or after execution. Each aspect is attached to a specific interception point (*IPoint*), which specifies the affected operation, the execution time (`pre` or `post`), and optionally a container (only possible for container-specific operations). Aspects provide arbitrary logic that may invoke CAPI operations and/or external methods. They can modify parameters (for pre-aspects) or return values (for post-aspects) and also raise errors that prevent the execution of the associated operation. For instance, a simple pre-aspect for replication could intercept write operations on a container and copy the included entries to remote containers. In addition, the aspect may inject certain meta data into the entries, like information about the replica locations.

Aspects must be designed with care, as errors may lead to deadlocks or prevent access to the space. In fact, aspects may be used to provide a simple access control mechanism by checking credentials included in the context parameter and granting access according to a fixed ACL. This solution does, however, not provide the flexibility of a solution based on declarative authorization policies, as specified in this work.

5.1.2 XVSM Middleware Runtime

The CAPI functionality can be realized using the XVSM runtime architecture, which is shown in Figure 5.2. It consists of several components that are themselves decoupled via special system containers with fixed coordination laws. Thus, a staged event-driven architecture [WCB01] is bootstrapped using own mechanisms. In a distributed environment, each space is represented by a separate runtime instance termed *XVSM core*. In order to participate in XVSM-based collaboration, any application or component has to instantiate its own core. Remote communication is realized by means of *sender* and *receiver* components, which enable transparent access to remote cores via the embedded CAPI component using a language-independent protocol.

XVSM operations are invoked by putting a corresponding *XVSM request* into the *XVSM request container*. Requests can be issued by local applications via the embedded

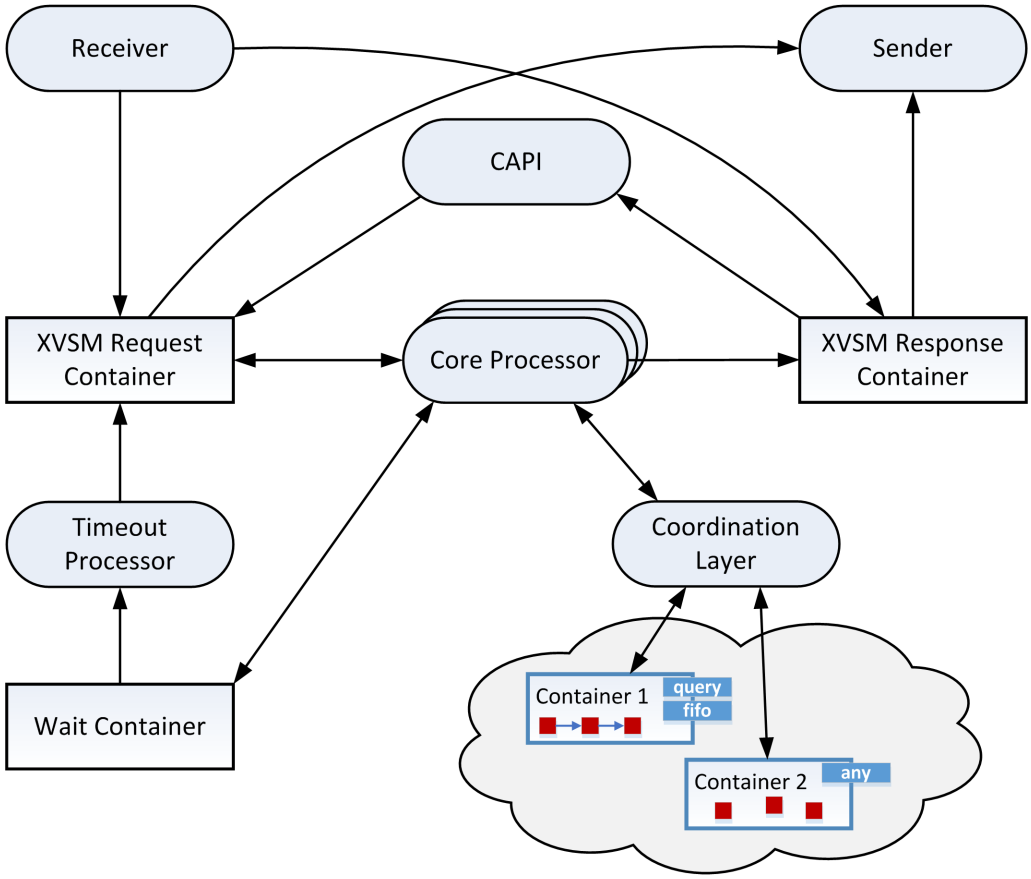

Figure 5.2: XVSM runtime overview

CAPI or they may be forwarded from remote cores via the receiver. Each request is represented by an entry that contains a unique *request ID*, the URI of the calling space instance (only for remote invocations), the operation name, and the specified parameters. After request processing, *XVSM responses* are written to the *XVSM response container*. They include the corresponding request ID, the invoker URI (if applicable), and the operation result (or an error message). If the request was performed on the local space, the response entry is directly taken by the CAPI and the extracted result is returned to the application. Otherwise, the sender component forwards it to the responsible remote XVSM core. When the CAPI issues a request that includes a remote space or container reference, the sender automatically forwards the request to the right XVSM core. The receiver then puts the subsequent response from this core into the local response container, from where the embedded CAPI can retrieve it.

The execution of requests is performed by one or more concurrently running *core processors*, which take an available entry from the request container, perform the corresponding operation (including potentially registered aspects), and write the response to the response container. For the actual space access, the *coordination layer* is invoked,

which realizes transactions and non-blocking container operations. The blocking behavior is enabled by the event processing logic of the core processor, which puts unsuccessful (but still valid) requests into a *wait container*. After processing a request, the core processor checks if waiting requests could potentially be satisfiable due to events caused by the executed operation (e.g., `take` requests are woken up by `write` operations on the same container). Matching requests are then rescheduled by writing them back into the request container. An additional *timeout processor* ensures that expired requests are removed in a timely manner from the wait stage, whereas the core processor generates a response with a suitable error message.

The modular architecture combined with the extensibility features of the CAPI support the definition of extension modules that can be largely bootstrapped via aspects, meta containers, and custom coordinators. Investigated extensions to the XVSM specification include asynchronous event notifications, lifecycle management, distributed container discovery, distributed transactions, replicated containers, and persistent container storage. An access control mechanism for XVSM can be integrated in a similar way, which is demonstrated in Section 5.3.

5.2 XVSM Access Control Model

The XVSM access control model has been designed according to the overall requirements specified in Chapter 4. Its main concept is the fine-grained protection of local containers using rule-based policies that are loosely based on XACML. Thus, an ABAC approach with expressive content- and context-based constraints is pursued, whereas the XACML policy syntax is greatly simplified and adapted to the needs of space-based middleware.

The model is independent from the used authentication mechanism. It assumes that each incoming request is properly authenticated and the corresponding subject attributes are included in a dedicated *subject property* that is part of the request context. Identity management is not part of the suggested approach. Instead, fully trusted identity providers are necessary that enable the management of users and their associated properties. For usage in practical scenarios, the access control model has to be combined with a secure communication infrastructure based on state-of-the-art encryption mechanisms that prevents eavesdropping as well as man-in-the-middle and replay attacks.

Like in XACML, an XVSM authorization policy consists of a set of uniquely named *rules* and a corresponding *combination algorithm*. Each rule specifies a *target*, a *scope*, and a *condition*, as well as an *effect* of either `PERMIT` or `DENY`. The target matches XVSM requests based on subject attributes, the accessed container, and the used operation. The scope restricts the entries for which the rule applies to a dynamic subset of the container based on the entries' application and coordination data. The condition imposes additional constraints based on the content of certain containers that hold entries depicting relevant context information. For any accessed entry, a rule only applies if the target matches the request, the condition is fulfilled, and the entry is within the scope. Conflicts due to overlapping rules with different effects are resolved using the configurable combination

algorithm. The following sections describe the policy language used for rule definition and the evaluation mechanism for determining an access decision based on these rules.

5.2.1 Policy Language

In contrast to the verbose policy definitions of XACML, the proposed access control model aims at simple and comprehensible rules that integrate existing XVSM concepts, like property sets and selectors. The following example, which permits write and take access for principals with role `User` on entries with type `UserData` in container *C1*, shows the basic structure of an XVSM access control rule:

RULE SimpleRule
SUBJECTS: [*role*: 'User']
RESOURCES: *C1*
ACTIONS: `write`, `take`
SCOPE: `type`(*UserData*)
CONDITION: -
EFFECT: `PERMIT`

More formally, the syntax can be specified using an EBNF-like notation:

⟨*Rule*⟩ = "`RULE`" ⟨*RuleID*⟩
"`SUBJECTS:`" ("`*`" | (⟨*SubjTmpl*⟩ { "`,`" ⟨*SubjTmpl*⟩ }))
"`RESOURCES:`" ("`*`" | (⟨*ContainerID*⟩ { "`,`" ⟨*ContainerID*⟩ }))
"`ACTIONS:`" ("`*`" | (⟨*AccessMode*⟩ { "`,`" ⟨*AccessMode*⟩ }))
"`SCOPE:`" ("`*`" | ⟨*Scope*⟩)
"`CONDITION:`" ("`-`" | ⟨*Condition*⟩)
"`EFFECT:`" ("`PERMIT`" | "`DENY`")

⟨*SubjTmpl*⟩ = "`[`" ⟨*PropPath*⟩ "`:`" ⟨*PropVal*⟩ { "`,`" ⟨*PropPath*⟩ "`:`" ⟨*PropVal*⟩ } "`]`"

⟨*AccessMode*⟩ = "`write`" | "`read`" | "`take`"

⟨*Scope*⟩ = ⟨*XQuery*⟩
| "`NOT`" ⟨*Scope*⟩
| ⟨*Scope*⟩ "`AND`" ⟨*Scope*⟩
| ⟨*Scope*⟩ "`OR`" ⟨*Scope*⟩
| "`(`" ⟨*Scope*⟩ "`)`"

⟨*Condition*⟩ = ⟨*ContainerID*⟩ "`|`" ⟨*XQuery*⟩
| "`NOT`" ⟨*Condition*⟩
| ⟨*Condition*⟩ "`AND`" ⟨*Condition*⟩
| ⟨*Condition*⟩ "`OR`" ⟨*Condition*⟩
| "`(`" ⟨*Condition*⟩ "`)`"

The complete syntax of all rule elements is defined in the appendix (see Section A.2.4). In the following, the individual rule components are described in detail.

Rule Target. The target of a rule is specified via three fields: *subjects*, *resources*, and *actions*. The `subjects` field defines a list of *subject templates* that represent affected users by means of a property set containing relevant security attributes, like a user ID, a role, or an organization. The responsible subject of an operation has to match at least one of these templates, otherwise the rule is not applicable. Template matching occurs similarly to the Linda-based selection already available in XVSM, i.e., each specified property in the template must correspond to an equivalent property in the subject information provided by the authentication mechanism. For instance, the subject template "[*role*: 'Prof', *affiliation*: 'TUWien']" is matched by the subject "[*userId*: 'Eva', *role*: 'Prof', *affiliation*: 'TUWien']", but not by "[*userId*: 'Stefan', *role*: 'Student', *affiliation*: 'TUWien']".

The `resources` field lists all containers for which the rule applies. As each XVSM core only controls access to its own containers, the full container reference is not required here. Instead, the local container ID can be used[3]. The affected operations are defined in the `actions` field, which can comprise any combination of `write`, `read`, and `take`, whereas take permissions implicitly include read permissions. Other CAPI operations are not included here as they can be protected in a bootstrapped way using the Secure Space architecture (cf. Section 5.3). General rules for all users, all containers, or all operations are supported via wildcards on the respective target fields.

Scope. The scope mechanism enables fine-grained, content-based access control by using XVSM queries (i.e., selector chains) for selecting the entries that are covered by the rule on the accessed container, thus exploiting the expressive and extensible coordination features of the middleware itself. The scope therefore defines a dynamic container partition that determines which entries are accessible by read or take operations and which entries can be written by write operations, respectively. For instance, all entries with a specific label or those matching a given template can be covered by a single rule. In the previously defined example rule, the scope "`type`(*UserData*)" ensures that user permissions are restricted to writing and retrieving entries with the specified type property.

The expressiveness may be additionally extended by combining multiple scope queries with logical operators[4]. Scope expressions joined with `OR` represent the union of the individual results, i.e., affected entries have to be matched by at least one of the specified queries. Similarly, `AND` forms an intersection (i.e., entries must be included in both scope parts), whereas `NOT` selects all entries in the container that are not matched by the scope expression. If access to the whole container shall be granted (or denied), a wildcard can be specified in the `scope` field.

[3] For non-anonymous containers, the container name provides another unique reference within the local space that can be easily mapped to the randomly generated container ID via the `lookupContainer` operation. For increased readability, self-explanatory container names will therefore be used for example rules in the course of this thesis.

[4] In previous publications, set operators where used instead to join the result sets of different scope queries (with equivalent semantics).

Condition. An optional condition facilitates context-based access control by using XVSM queries in a similar way. It consists of one or more condition predicates that are combined via logical operators. Each predicate specifies a local container and an associated XVSM query. The predicate evaluates to true if the application of the query on the specified container is successful and returns at least one entry. The rule is only applicable if the combination of all predicate results yields true.

In contrast to the scope mechanism, which defines a dynamic container partition that can be accessed by the subject, conditions enable constraints on additional context data that is not directly written or queried by the checked operation. This mechanism can be used for rules that depend on the existence (or absence) of certain entries in a specific container (e.g., based on their type) or on property values within these entries (e.g., using a `query` selector). Context information can be represented by different entries in arbitrary containers, which are modified via local or remote processes. Relevant context information may include the current application state, dynamic configuration settings, or sensor measurements. For instance, the condition "*ContextC* | `key`('app1') | `query`(*state* = 'active')" requires a specific application to be active, which is indicated by an entry in a dedicated context container (*ContextC*) with key "app1" and a corresponding `state` property. Processes that change such context entries may thus influence permissions of other users. To prevent misuse, access to these context entries has to be secured via additional rules that only permit modifications by privileged subjects.

The expressiveness of the condition mechanism may be enhanced via aspects that dynamically generate context entries based on computations that cannot be expressed using regular condition predicates. For instance, aspects could perform comparisons of multiple entries in different containers, generate aggregated values, or count the number of access attempts. They can then write a corresponding context entry to a container, where it can be subsequently checked by a condition. However, in general the usage of such extended conditions is not recommended because they are not defined within the authorization policy, which leads to access control semantics that may be difficult to understand.

Dynamic Parameters. Besides the context represented by entries in the space, an access decision may also be affected by properties stored in the request context. This includes the authenticated subject attributes as well as additional properties included by the client or the space runtime (e.g., timestamps). In order to support fully context-aware rules, *dynamic parameters* are introduced for XVSM queries within `scope` and `condition` fields. These parameters are referenced with a "$" sign, followed by the distinct name of the corresponding request context property. They may replace any regular parameter within a selector as long as the data types are compatible. Before the scope and condition of a rule are evaluated, all dynamic parameters get resolved and are replaced by the corresponding values from the request context. For instance, the selector "`label`($*subject.userId*)" matches entries that are associated to the invoking user. Within a scope query, this selector ensures that each user can only access his or

her own entries. In a similar way, a condition may specify that a registration entry must exist for any valid user.

Some rules may solely depend on request context properties instead of specific entries within a container. A query like "`query`($*time* ≥ 08:00 ∧ $*time* ≤ 17:00)" would match any entry as long as the request was issued during business hours. Thus, time-specific permissions can be issued via a condition that uses this query and targets an arbitrary container with at least one entry.

Example Rules. The policy language supports different types of rules that exceed the expressiveness of simple ACLs. The following two examples outline possible constraints that are supported by this approach:

RULE R1
SUBJECTS: [*role*: 'Clerk', *affiliation*: 'A'], [*affiliation*: 'B']
RESOURCES: *C1*
ACTIONS: `read`
SCOPE: `label`('public') `OR` `label`($*subject.userId*)
CONDITION: -
EFFECT: `PERMIT`

RULE R2
SUBJECTS: [*role*: 'Manager', *affiliation*: 'A']
RESOURCES: *C1*, *C2*, *C3*
ACTIONS: *
SCOPE: *
CONDITION: `NOT` *LockC* | `label`('lock')
EFFECT: `PERMIT`

Rule `R1` permits read-only access to container *C1* for subjects with role `Clerk` from company `A` as well as any user from company `B`. However, only specific entries are accessible, which must be either marked as public or specifically assigned to the requestor. Rule `R2` grants unrestricted access to three containers for managers from `A`, unless a lock entry exists in a specific container (*LockC*), which temporarily deactivates this permission.

5.2.2 Policy Evaluation

Figure 5.3 shows the necessary evaluation steps for authorizing a take operation based on a rule inspired by the firewall management scenario. This rule grants limited take permissions to subjects with role `EventManager` on an event container. Each request object for a container access operation is compared with the targets of available rules (step 1). Only when all three target fields match, the rest of the rule is evaluated. In the second step, the condition is analyzed, which induces queries to one or more context containers. In the example, the rule only applies when a specific token entry (with

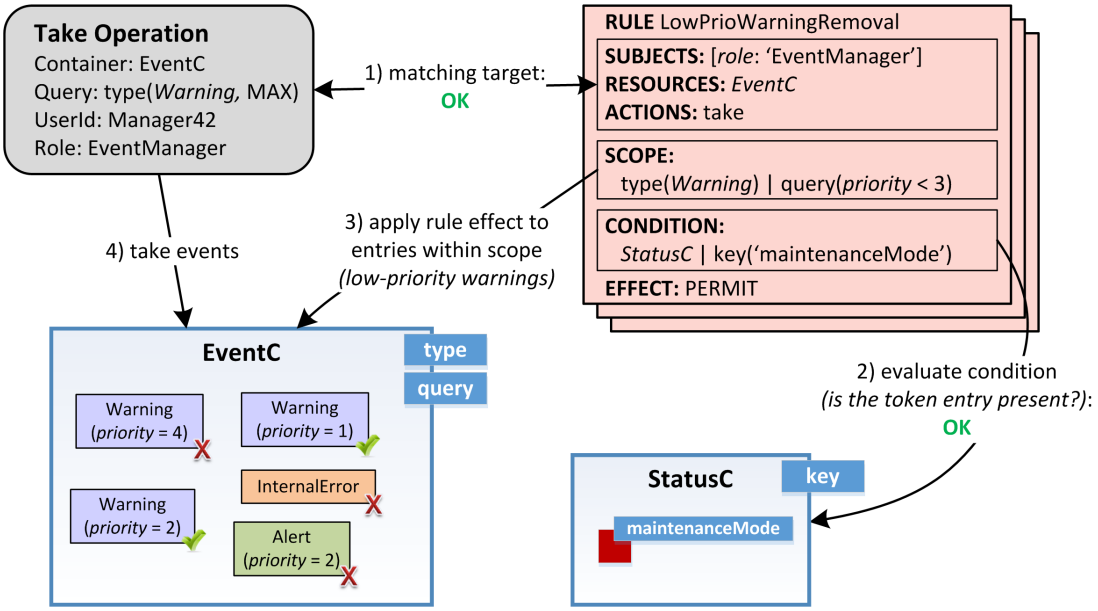

Figure 5.3: Rule evaluation semantics for XVSM access control model (based on [CDJ+13])

key "maintenanceMode") is present in the status container, thus indicating that manual removal of events is currently enabled. An administrator with appropriate permissions on this status container can remove this entry at any time in order to deactivate the rule without actually changing the authorization policy. Afterwards, the rule scope is computed (step 3), which determines a container subset that consists of entries with type `Warning` and a `priority` property with a value of less than three. In the depicted state of the event container, this rule therefore permits access to two of the five entries. For `read` and `take`, the scope is dynamically computed on the current state of the addressed container before the actual operation is performed. This enables coordinators to ignore denied entries and select accessible alternatives instead, similar to the treatment of entries locked by transactions. Finally, the take operation executes (step 4) and removes matching entries. Due to the used count parameter, all accessible entries (i.e., the two permitted warnings) are returned.

For `write`, it has to be checked if the given entries are within the scope. As XVSM queries only operate on containers and not on individual entries, a tentative container state is examined that already includes the entries to be written. If access is denied, the effects of this write operation have to be reversed before they become visible. Due to this approach, fine-grained access control is also possible for write operations.

If more than one rule applies for a given space invocation, a specifiable combination algorithm has to be used, which determines the evaluation order of the rules and the resolving strategy for conflicting decisions. In contrast to XACML, access decisions are not combined at the request level but separately for each entry. This is due to the possibility for dynamic scopes and the bulk operations supported by the CAPI, where

multiple entries can be accessed by a single operation. In some cases, there may be no single rule that grants privileges for all involved entries, thus a request-level combination would have to deny access. However, there may be multiple rules that each permit access to a different set of entries in the container. As long as each involved entry is covered by at least one of these rules, access should be granted.

The combination algorithm is responsible for computing an access decision for each entry of the accessed container. The following decision values are possible:

- **PERMIT:** Access to this entry is granted.
- **DENY:** Access to this entry is explicitly denied.
- **NOT_APPLICABLE:** None of the evaluated rules was applicable for this entry.
- **INDETERMINATE:** An evaluation error occurred for a rule that could have been applicable for this entry.

An operation is only allowed if all queried or written entries have an access decision of `PERMIT`. Combination algorithms need to have clear semantics in order to allow for comprehensible and manageable policies. Possible strategies include:

- **PERMIT-OVERRIDES:** If at least one rule permits access to an entry, the decision is `PERMIT`.
- **DENY-OVERRIDES:** If at least one rule denies access to an entry, the decision is `DENY`.
- **FIRST-APPLICABLE:** A fixed rule order is used and the first applicable rule for an entry defines the decision.

XACML supports further structuring of an authorization policy via nested policy sets that group rules according to their target and combination algorithm. This introduces levels of indirection that may reduce comprehensibility and scalability. For XVSM, a more lightweight approach is chosen with a single policy that contains all rules, although extensions involving nested or possibly even distributed policies are conceivable if the need for highly structured policies arises (e.g., for very large projects with multiple security domains per space). Despite the simple policy structure, the policy language is still able to express most of the relevant security constraints on space containers via a minimal number of rules.

5.3 Secure Space Architecture

In order to protect XVSM spaces from unauthorized access, a suitable access control mechanism has to be incorporated into the middleware runtime that enforces the specified authorization policies and supports their management. Authorization must not only

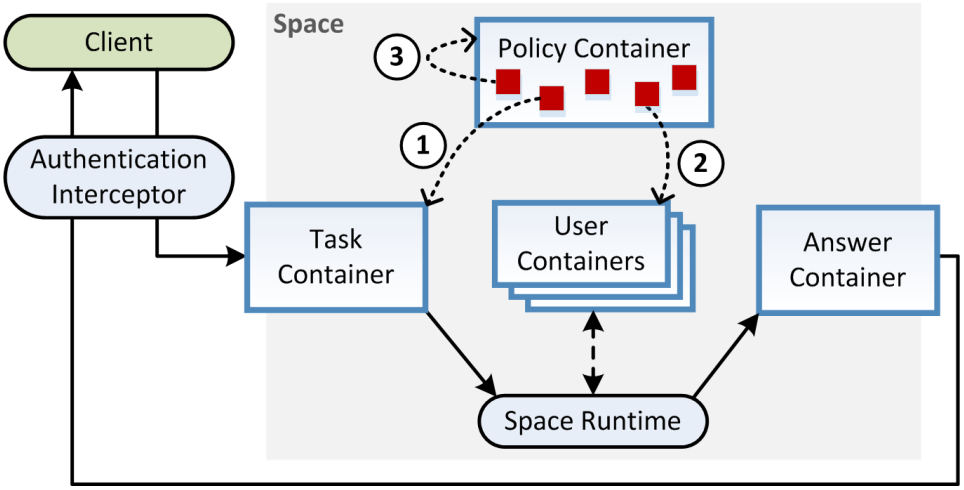

Figure 5.4: Generic Secure Space architecture

cover container access, but all CAPI operations. Otherwise, access control could be bypassed via aspects or by deleting containers. According to the bootstrapping principle, large parts of the functionality can be realized with own mechanisms.

The authorization architecture is based on the generic concept of a *Secure Space*, which is depicted in Figure 5.4. Clients can only indirectly interact with the space by means of two meta containers. They write requests into the *task container*, which are then executed by the space runtime (possibly involving a user container). Afterwards, the result can be retrieved from the *answer container*.

The Secure Space architecture assumes that each client includes credentials linked with the identity of its user when accessing the space. This may be a certificate, an SSO token, a password, or any other proof of identity. An exchangeable authentication interceptor, which is configured according to the supported authentication mechanism(s) of the application scenario, verifies these credentials and injects the authenticated subject attributes into the request. These attributes are then examined during the policy evaluation process to determine the permissions of the subject.

The authorization policy is stored in a separate *policy container*, where each rule is represented by a single entry. This approach enables the definition of different types of rules (depicted by the numbered arrows in Figure 5.4), as the policy language can reference meta containers just like regular user containers. *Invocation-based rules* (1) define which operations a subject is able to perform on the space by restricting the write operation on the task container. Thus, access depends on the request type or on specific parameters. For instance, users may be permitted to perform container access, container lookup, and transaction handling, but no container and aspect management operations. *Data-based rules* (2) operate on regular (non-meta) containers whose names and contents depend on the application scenario. They define which users may write, read, or take which data in these containers. *Administrative rules* (3) define permissions on the policy container itself, thus enabling the management of administrator privileges using the

same concepts as for regular user permissions. Users with the necessary permissions can simply modify the policy container via the space runtime using ordinary container access operations. To enable the initial definition of permissions, a special administrator user with full privileges must exist. Restricted delegation of rights is possible as the scope mechanism may be used to authorize subjects to write or remove only certain kinds of rules (e.g., involving a specific container). The following two rules show simple examples for invocation-based and administrative rules, respectively:

RULE CreateContainerRule
SUBJECTS: [*role*:'Manager']
RESOURCES: *TaskContainer*
ACTIONS: `write`
SCOPE: `type`(*CreateContainer*)
CONDITION: -
EFFECT: `PERMIT`

RULE AdminRule
SUBJECTS: [*role*:'Admin']
RESOURCES: *PolicyContainer*
ACTIONS: *
SCOPE: *
CONDITION: -
EFFECT: `PERMIT`

The first rule allows managers to invoke the `createContainer` CAPI operation, whereas the second rule delegates administrator privileges to all subjects with a corresponding role. Due to this approach, a dual protection layer is established for container access operations (including policy administration). Users must be authorized to access the space with the requested operation (`write`, `read`, or `take`) in the first place. During execution of the container access, also the data-based (or administrative) rules must be satisfied. It is also possible to define rules on the answer container, which may be used to ensure that malicious users cannot eavesdrop or delete responses for other clients.

In the following section, the integration of the generic Secure Space architecture into the XVSM middleware runtime (cf. Section 5.1.2) is described.

5.3.1 Integration into XVSM Runtime Architecture

The concepts of the Secure Space can be mapped to already existing mechanisms of the XVSM runtime. By design, the application hosting the space has full control over its local XVSM core. Therefore, access control is only enforced for requests coming from remote instances.

The task container corresponds to a partition of the XVSM request container that only contains remote entries, which have entered the system via the receiver component. When such entries are rescheduled, their authorization has to be checked again as the policy may have been changed in the meantime or modified container states may affect applicable rules due to the dynamically evaluated `scope` and `condition` fields. Requests from the local CAPI, on the other hand, are implicitly permitted. The space runtime consists of the core processors and adjacent components for realizing the coordination logic and the blocking behavior. The answer container can be associated with the XVSM response container, although the latter is not directly accessed by remote clients. Instead, the response is automatically pushed to the invoking core, which can be interpreted as an implicit blocking take operation caused by the request. As unauthorized access by third

parties is thus prevented and content- or context-based restrictions are already covered by the other rule types, authorization rules on the response container are not necessary. However, besides the previously explained synchronous CAPI invocation mode, also asynchronous variants of the XVSM operations may be supported. In this case, users must include a reference to an arbitrary container that should act as an explicit answer container. The invoker can then retrieve the result via a regular take operation at a later time. Therefore, this container must be secured with appropriate rules so that only the original invoker can access the entry.

Unlike request and response containers, the policy container is a regular XVSM container that can be accessed using the CAPI. The local administrator user corresponds to any principal that controls the hosting application. As local access via the embedded CAPI is implicitly permitted, the initial permissions can be set after the space is created by writing rule entries to the policy container. By default, no external access is allowed as long as the policy container remains empty.

In the following, the architecture for integrating the Secure Space concept with XVSM is described, which is largely bootstrapped using existing middleware features like containers, coordinators, transactions, and aspects.

Access Control Architecture Overview

The space runtime is responsible for enforcing the specified authorization policy. Simple container-based ACLs could be realized by intercepting requests before executing a CAPI operation and matching them against the rule set, while entry-specific ACLs may be enforced before returning results to the user. However, the dynamic scope and condition mechanisms of the XVSM access control model necessitate a tighter integration with the coordination layer and its query capabilities.

Figure 5.5 shows the access control architecture of XVSM, which extends the existing coordination layer with an *access manager* component that enforces authorization according to the active policy. A bootstrapped approach involving XVSM queries is applied in order to match rule targets, check conditions, and evaluate scopes.

Remote requests, which are distinguished from embedded requests via a context flag set by the runtime, need to be authenticated and authorized. This is supported via two aspects that are registered to run before each CAPI operation, which is realized by adding them to all "`pre`" interception points. The *authentication aspect* (`AuthN`) acts as a configurable authentication interceptor that verifies the claimed identity of the invoker. In order to avoid the need for a separate authorization mechanism for the request container, which is not managed by the coordination layer, the *request authorization aspect* (`ReqAuthZ`) checks permissions for writing requests in a bootstrapped way using a separate meta container. The aspect mechanism can also be used to protect the access control architecture itself by preventing the deletion of relevant aspects and meta containers.

For every container access operation, the core processor invokes the coordination layer with the parameters specified in the corresponding request. To support authorization, the coordination layer forwards authenticated requests to the access manager, which

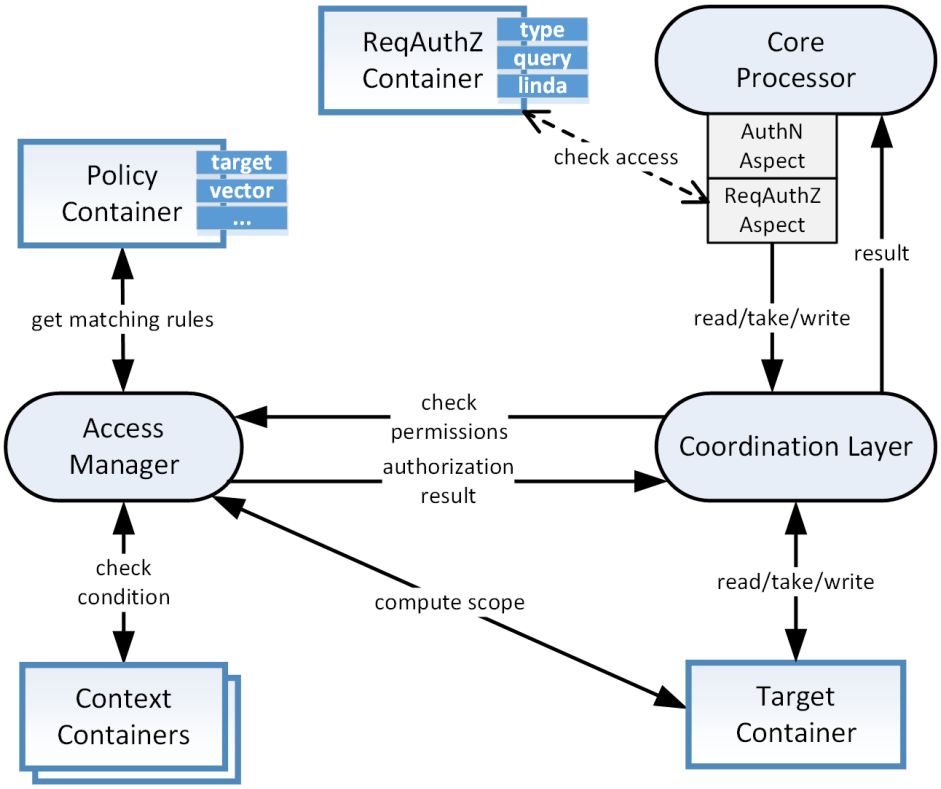

Figure 5.5: XVSM access control architecture

evaluates each request according to the current policy and its configured combination algorithm. The returned authorization result has to be enforced by the coordination layer for the corresponding container access operation. The actual write or query logic is then performed by accessing the container targeted by the request, which includes calls to the registered coordinators and a transaction manager component that realizes the locking behavior (not shown in the figure).

In the terminology of XACML, the coordination layer acts as PEP, while the access manager represents the PDP. These components could also be decoupled via an intermediate container (similar to xDUCON). However, as their communication occurs synchronously and no other components are directly involved, this would merely increase the overhead of the authorization mechanism. Instead, both the coordination layer and the access manager are invoked in the context of the core processor (i.e., in the same thread). Concurrency is ensured at the request level by means of multiple core processors that execute requests in parallel. Both PAP and PIP are bootstrapped with own mechanisms using regular XVSM containers. The PAP is represented by the policy container, which enables the management of rules by administrators and their retrieval by the PDP, whereas all containers invoked by the PDP during the evaluation of scope and condition queries form the PIP.

In the following, the relevant components and their interactions are described in more detail.

Request Authentication

The authentication aspect is responsible for verifying the identity of the invoking principal and setting the security attributes in the request accordingly. It retrieves the claimed security attributes (e.g., the user name and a role) and the included credentials (e.g., a password) from the `subject` and `creds` properties in the request context, respectively. Depending on the used authentication mechanism, checking this information may involve communication with external identity providers or access to local databases. If the authentication succeeds, the credentials are removed from the request context, while the nested subject property may be enriched with additional information about the principal retrieved during the authentication process. The included attributes depend on the responsible identity provider, but in most scenarios, at least the fields `userId`, `role`, and `affiliation` (i.e., the subject's organization) should be supported. Besides these verified security attributes, the invoker may also provide additional subject information within a special `EA` property for extra attributes that are not authenticated. Such attributes are treated as additional information for which the authenticated principal vouches. Accordingly, they can also be addressed in subject templates but have to be used with care, as they cannot be verified by the local runtime. If a request cannot be authenticated, the aspect indicates an error, which causes the core processor to cancel the operation and return an authentication error to the client.

Authorization Workflow

The executed authorization workflows are shown in Figure 5.6 (for query operations) and Figure 5.7 (for write operations). Both variants use the same mechanism for computing the authorization result, but enforce it in different ways. As already mentioned, authorization for read and take operations is performed before the query is executed on the target container, whereas write operations are checked after the entries have been written. The atomicity of these workflows is ensured via the transaction mechanism. In the following, the relevant steps are explained.

Computing the Authorization Result. To retrieve matching rules in an efficient way, the access manager first performs a non-blocking read operation on the predefined policy container, which is managed by a special `target` coordinator. Its selector requires the subject property, the operation type, the reference of the targeted container, and a count parameter (typically `ALL` to retrieve all matching rules at once). As a configurable rule order may be relevant for some combination algorithms (e.g., `FIRST-APPLICABLE`), an additional `vector` coordinator is supported.

For each matching rule, the `condition` and `scope` fields are evaluated in a similar way by invoking non-blocking read operations directly on the respective containers. A condition induces one or more queries on the specified context containers, whereas a

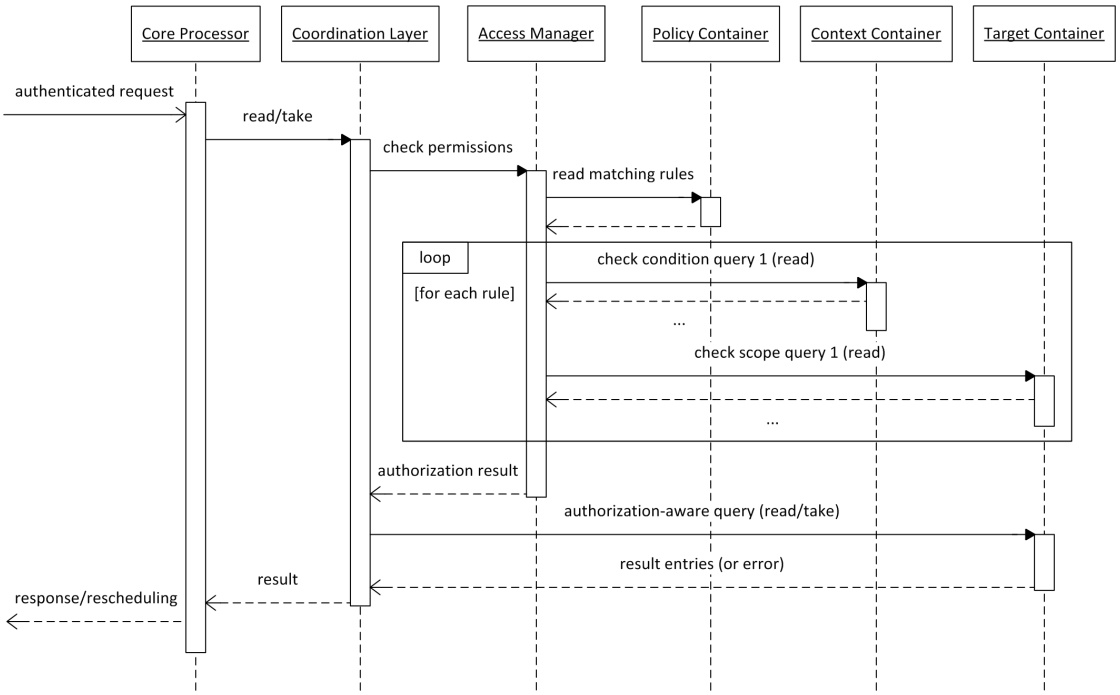

Figure 5.6: XVSM authorization workflow for query operations

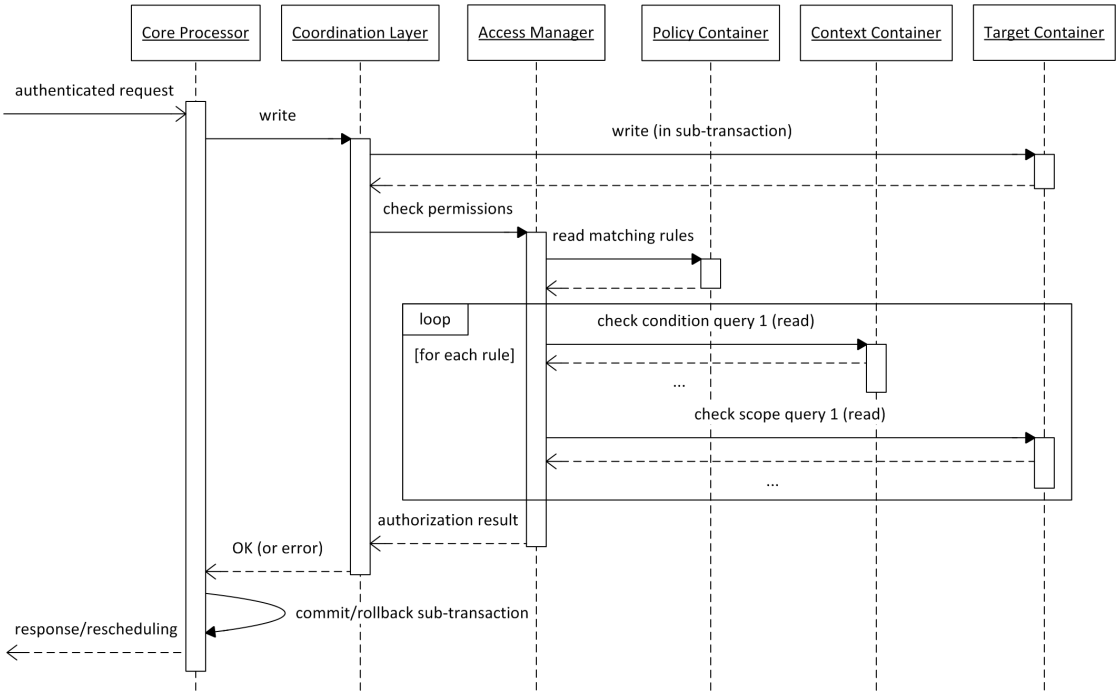

Figure 5.7: XVSM authorization workflow for write operations

scope is determined via one or more queries on the target container. Dynamic parameters in the `scope` and `condition` fields are replaced with their corresponding values in the request context before issuing these queries.

As separate access decisions are required for each entry, a *decision mapping* is created that links entries to their corresponding decisions. Initially, each entry of the target container is assigned to the decision value `NOT_APPLICABLE`. When a rule applies, its effect (`PERMIT` or `DENY`) is stored for all entries within the scope. During the evaluation of the rules, the access manager then iteratively updates this mapping until all rules are processed. In the case of conflicting rules, the combination algorithm has to decide whether a decision that was set by a previously evaluated rule should be kept or replaced. In many cases, an access decision applies for the whole container, e.g., when no scope is defined or no matching rules are found. It is therefore not necessary to build an access decision map, which may be a costly operation for containers with many entries. Instead, a *general decision* object is returned that contains a single decision value for the entire operation. As an optimization, such overall decision values can also be combined with a decision mapping in order to provide a default decision for entries that are not explicitly specified in the mapping. In case of errors during the evaluation process that may affect the result (e.g., unresolvable dynamic parameters, failed condition or scope queries due to missing containers or invalid parameters), a general decision of `INDETERMINATE` is used. As an example for a possible implementation of this step, the `DENY-OVERRIDES` combination algorithm is specified in the appendix in Section B.1.

Enforcing the Authorization Result. After the access decision has been computed, it has to be enforced by the coordination layer. In case of a general decision of `PERMIT`, the operation is authorized and no further checks are necessary. If there is a general decision with a different value, the request is canceled.

If the authorization result corresponds to a decision mapping, the behavior depends on the operation. For `read` and `take`, an *authorization-aware query* is issued, which includes the decision mapping as an additional argument for the query mechanism of the target container. In the existing XVSM model, coordinators can already check if certain entries are locked by another transaction within their `select` method. Using the decision mapping, they can now also determine if the entry is accessible by the subject. Depending on the coordinator logic, they may choose to skip the entry in favor of a permitted one, or return an error if this is not possible. The possibility to select alternative entries (if available) is useful for non-deterministic coordinators like `linda` or `type` to avoid unnecessary access violations. If chained selectors are used, only the first coordinator has to examine the decision mapping, as the others operate on the result entries of the first stage, which are already authorized.

For `write`, the actual container operation is not influenced by the authorization mechanism, as it is already performed before the access manager gets invoked. For enforcing the authorization result, the transaction mechanism of XVSM is exploited instead. Each request execution by the core processor is performed in an internal sub-transaction, which is only committed when the operation completes successfully. As

all described container accesses are part of this sub-transaction, this also ensures that the access manager operates on a consistent state. The coordination layer checks if all written entries are permitted according to the decision map (or the general decision). Otherwise, it returns an error to the core processor, which immediately rollbacks the sub-transaction so that the denied write operation never becomes visible.

When an authorization error occurs, the core processor has to decide how to proceed with the request. Usually, it generates a response with an appropriate error message for the invoking client. However, a different strategy is applied for authorization-aware queries that could not be fulfilled because not enough matching entries were accessible by the subject. Depending on the used selectors, such operations may succeed in the future even if the policy does not change, as new entries that match both the query and the scope of applicable rules could be written to the container. Therefore, these requests are put into the wait container for rescheduling at a later time as long as their timeouts have not yet expired. This approach enables transparent access to containers that are partially accessible for the subject. Denied entries are simply ignored and the CAPI blocks (or returns a timeout error), as if the searched entries do not exist. In contrast to locked entries, such entries are also skipped when using the count parameter `ALL`. As a consequence, covert channels are prevented where users could learn about the existence of denied entries with certain properties by checking whether an authorization error is returned or a regular timeout occurs. For write operations and queries rejected based on a general decision, rescheduling would not be effective unless the policy (or the context) changes in the meantime, which is not very probable in most scenarios. Instead, it is more reasonable to immediately inform the client about the denied access, so that appropriate measures can be adopted (e.g., contacting the administrator). Clients should not be able to retrieve secret information in this way[5].

Authorization for Invocation-Based Rules

The request authorization aspect is invoked after the authentication aspect and thus intercepts all authenticated requests. Invocation-based rules are enforced with own mechanisms by trying to write the request into a special request authorization container that represents the task container of the Secure Space architecture. The aspect performs a regular write operation via the embedded CAPI (evaluated by a concurrently running core processor) while including the subject property of the invoking client in the request context. As this is an internal access, which is normally implicitly allowed, a special flag has to be set that forces authorization on target containers, while bypassing the authentication and request authorization aspects to avoid endless recursion.

[5]Coordinator-specific exceptions (e.g., due to duplicate keys) and locking constraints may cause limited information leaks for write operations, as these conditions are checked before authorization. In such cases, a generic response has to be sent to the client that does not disclose why an entry could not be written. Users may also learn indirectly about certain container properties (e.g., locked coordinators or full containers) when their write operation blocks instead of immediately returning with an authorization error. To prevent this, non-zero timeout parameters could be disallowed for remote write operations, which may be realized via an invocation-based rule.

The access manager authorizes this operation based on the stored permissions for the subject on the request container. To enable high expressiveness via fine-grained scope constraints, different coordinators may be supported on this container, including `type`, `linda`, and `query`. If the write operation is successful, the request is authorized and the written entry can be removed. Otherwise, the aspect forwards the authorization error to the core processor, which returns a corresponding error message to the client.

Compared to the original Secure Space architecture, the requests are not authorized when they are received, but when they are actually processed. From a security standpoint, this approach is actually preferable, as all types of authorization rules are enforced whenever a request gets rescheduled. Thus, permissions for previously issued blocking operations could be withdrawn dynamically using invocation-based rules.

Policy Administration

The policy container has a preconfigured name that enables each authorized user to retrieve its container reference and access the included rules. To create a rule entry, the rule fields can be mapped to corresponding properties, which are specified in Section A.2.5 of the appendix. Besides the already mentioned `target` and `vector` coordinators, additional coordination laws may be used in order to ease policy management. The `key` coordinator ensures uniqueness of rule IDs and enables the update and deletion of specific rules, whereas `any` or `query` coordinators provide an overview of currently active rules.

5.4 Secure Service Space: Towards a Workflow Model

The Secure Space already enables secure data-driven coordination of autonomous components in a distributed environment, but it does not consider the possibility to encapsulate coordination logic into reusable services that can be invoked by remote clients. In Section 2.2.2, it has been shown that the decoupled interaction style of tuple spaces is well-suited for realizing service frameworks and workflow models. Thus, service invocation can be defined as a coordination pattern on top of XVSM: Services wait for specific request entries in a *service request container* (using a blocking take operation), perform their internal service logic, and finally write their result into a *service response container*, from where it can be retrieved by the invoker in a synchronous or asynchronous way. This pattern is commonly used for space-based service calls, e.g., in the Secure Lime implementation of SMEPP [BBB+08] and also in the XVSM runtime architecture itself (cf. Figure 5.2), where the core processor represents a meta-level service that executes CAPI operations.

In order to realize their behavior, services may need to store and query user-specific data, update their internal state, or coordinate themselves with other components (local or remote). *Space-based services* access XVSM containers also for these tasks, which minimizes complexity by using already available middleware functionality. Such a service-based approach has several advantages compared to direct container access by clients:

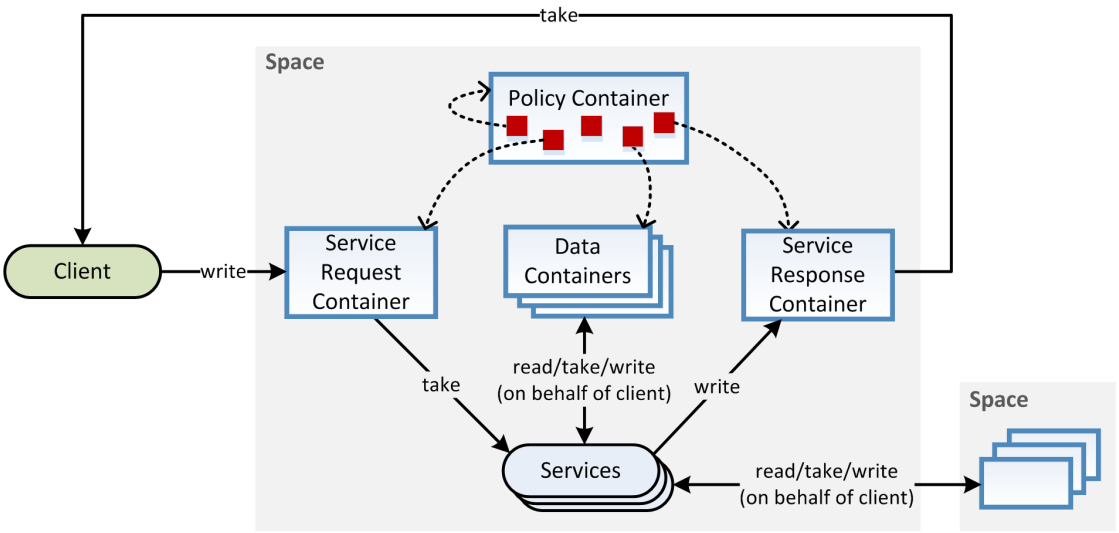

Figure 5.8: Secure Service Space architecture

- Generic coordination logic may be encapsulated by proven services that can be reused by different applications, which would otherwise require time-consuming and error-prone development efforts.
- Maintainability is improved, as the containers and the corresponding coordination logic are managed by a central entity, i.e., the application hosting the space. Modifications like changed entry formats or coordinators do not require changes in the distributed clients as long as the service interface (i.e., the request entry format) stays the same.
- Data entries are accessed by local services instead of remote clients. This reduces network traffic and latency, as only the final response of the service is sent back to the client instead of all intermediate results.
- Internal data structures and coordination mechanisms are hidden by the services. Thus, accidental and intentional misuses are prevented.

A suitable access control mechanism for this space-based service approach must protect the service request and response containers and prevent direct manipulation of internal data containers. However, permissions on the data containers may still depend on the identity of the invoker, even though the corresponding CAPI operations are performed by internal services. For instance, a service should not be able to provide personal data of other users to the invoker. Such restrictions could be enforced directly in the service code, but this would violate the separation of concerns between coordination and security. Thus, a delegation mechanism is required for services that act on behalf of a user to access local or remote containers.

A secure service framework that fulfills these requirements can be bootstrapped on top of the Secure Space. This *Secure Service Space* architecture is depicted in Figure 5.8. It resembles the Secure Space concept, but operates on a higher layer (i.e., application level instead of middleware runtime). Clients and services interact with the XVSM containers using regular CAPI operations. Access to these containers is controlled via data-based rules in the policy container, which can be categorized into three sub types. Firstly, service invocation is regulated via permissions on the service request container (e.g., based on the request type). Secondly, the indirect access to local or remote data containers via services that act on behalf of the user is controlled. For indirect access on remote spaces, also the service itself has to be authorized at the target space. Thus, the trust level among distributed service frameworks can be specified via authorization rules. Thirdly, also the service response container has to be protected so that only the request invoker can retrieve the corresponding response. Additionally, invocation-based rules on the XVSM request container still apply, even though this is not explicitly shown in the figure. These rules ensure that the aforementioned restrictions cannot be bypassed by denying aspect and container management operations for remote users that are not administrators of the space. This extended security architecture enables different forms of access control, which can be selected according to the application scenario:

- **Data-centric authorization:** Access control determines the supported operations on data containers for each user. The invocation of services is not restricted, but container access performed on behalf of the client has to be authorized. Thus, fine-grained permissions for indirect container access can be specified that do not require knowledge about the service logic. In some cases, it may also be beneficial to circumvent the service framework and permit direct access to certain data containers for remote users, e.g., for ad-hoc collaboration among peers without the need for installing a service or for computationally intensive data processing tasks that can be executed more efficiently in a distributed fashion.
- **Service-centric authorization:** Service invocation is authorized based on the request type and possibly also certain parameters. Indirect access to data containers via services is always allowed, while direct container access from external sources is denied. This allows for simple policies that links subjects to permitted tasks.
- **Combined authorization:** Both approaches can be combined to form highly expressive and flexible policies. Basic permissions are set on the service level, which can then be refined based on the containers and entries that may be accessed on behalf of the client. Thus, an additional layer of protection is created. Users can only invoke services for which they are authorized, whereas these services are automatically constrained by the rights of their invokers.

5.4.1 Bootstrapping of the Secure Service Space

The Secure Service Space can be implemented with already existing mechanisms of the XVSM-based Secure Space, as described in the following.

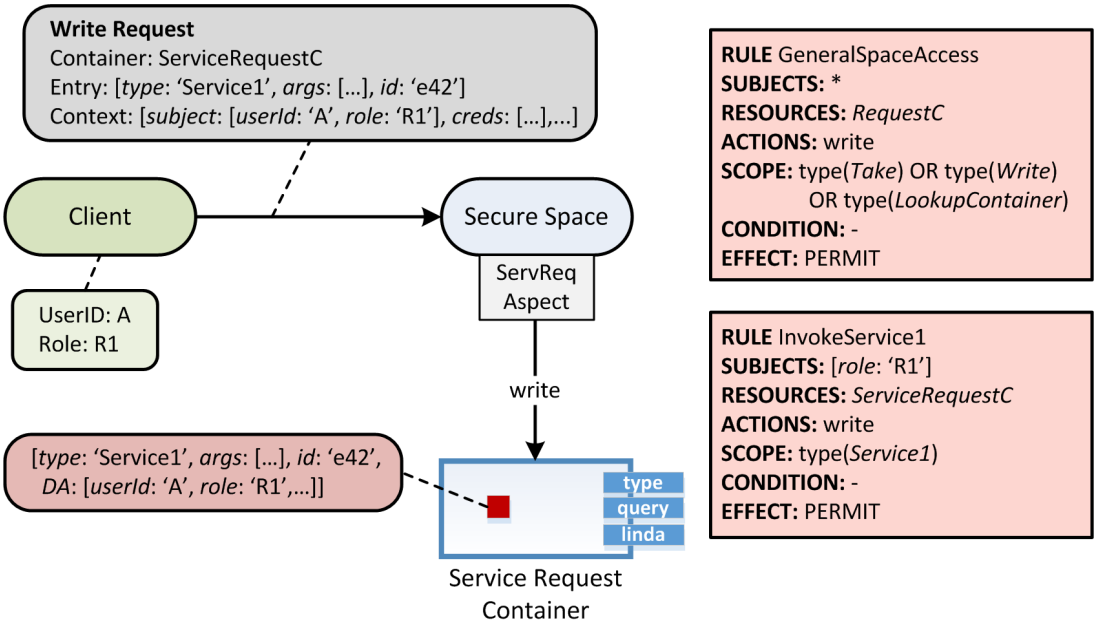

Figure 5.9: Service invocation in the Secure Service Space (with example rules)

Service Invocation

Figure 5.9 depicts the service invocation process and related authorization rules. To write a request for `Service1` into the service request container, a client associated to user *A* issues a corresponding XVSM request to the middleware runtime, which authenticates and authorizes the operation using the mechanisms described in Section 5.3.1. Each service request entry contains the request type (`type`), a service-specific property with parameter values (`args`), and a unique service request ID (`id`).

The first rule (`GeneralSpaceAccess`) grants all authenticated subjects the rights to invoke write, take, and container lookup operations on the Secure Space, which is required to use the service framework. However, by default no containers are accessible. The second rule (`InvokeService1`) explicitly permits subjects with role `R1` to write request entries with type `Service1` into the service request container.

In order to enable delegation support, an additional service request (`ServReq`) aspect is installed for the pre-write interception point on the service request container. It executes after the existing access control aspects and attaches the subject's security attributes from the context to the written entry within the delegated attributes property (`DA`), so that the identity of the invoker can be reused in future operations.

The service framework is a server application that initializes a Secure Space and then starts the installed services within separate threads to allow for concurrency. Idle services wait for new requests using blocking take operations with corresponding `type` selectors. Like any local access via the embedded CAPI, this operation is implicitly authorized.

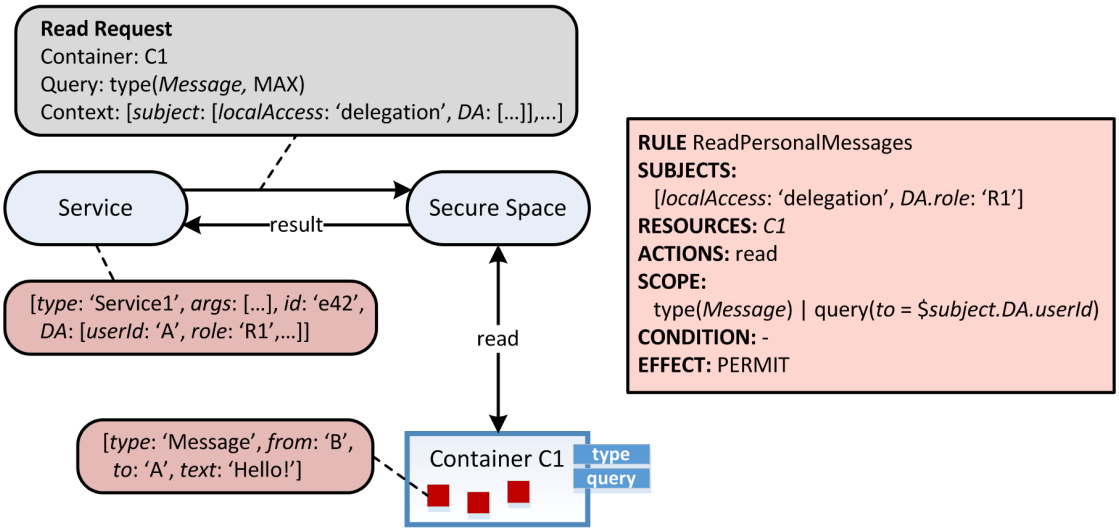

Figure 5.10: Indirect container access (local) via space-based service (with example rule)

Indirect Container Access

Figure 5.10 shows an indirect access to a local container by a service on behalf of the user. In the example, the service reads all accessible messages, which are restricted by a rule that only permits messages where the recipient field (`to`) corresponds to the ID of the invoking user. In order to distinguish delegated access from direct access by external users, a special subject structure is chosen for the request context, which also has to be considered for the respective authorization rules.

For *indirect local access*, the `DA` property from the request entry is included in the subject together with an additional `localAccess` property with value "delegation", which indicates the delegated access mode. The authorization of data-based rules on the local space is forced using the same request context flag as for the bootstrapped evaluation of invocation-based rules in the request authorization aspect of the XVSM access control architecture.

For *indirect remote access*, both the service and the invoking user have to be authorized. Instead of the `localAccess` property, the service has to include its own security attributes as well as credentials to back them up, which may be configurable via the service framework. As the delegated attributes cannot be directly verified at the target space, the `DA` property has to be nested within the `EA` property for extra, non-authenticated attributes.

This approach facilitates a clear distinction between direct, indirect local, and indirect remote access within subject templates of authorization rules. For direct access rules, subject templates do not contain delegated attributes (with prefix "`DA.`" or "`EA.DA.`"). As shown in Figure 5.10, subject templates for indirect local access rules consist of the mandatory `localAccess` field together with one or more delegated attributes. If the target container were on a remote space instead, the subject template for a

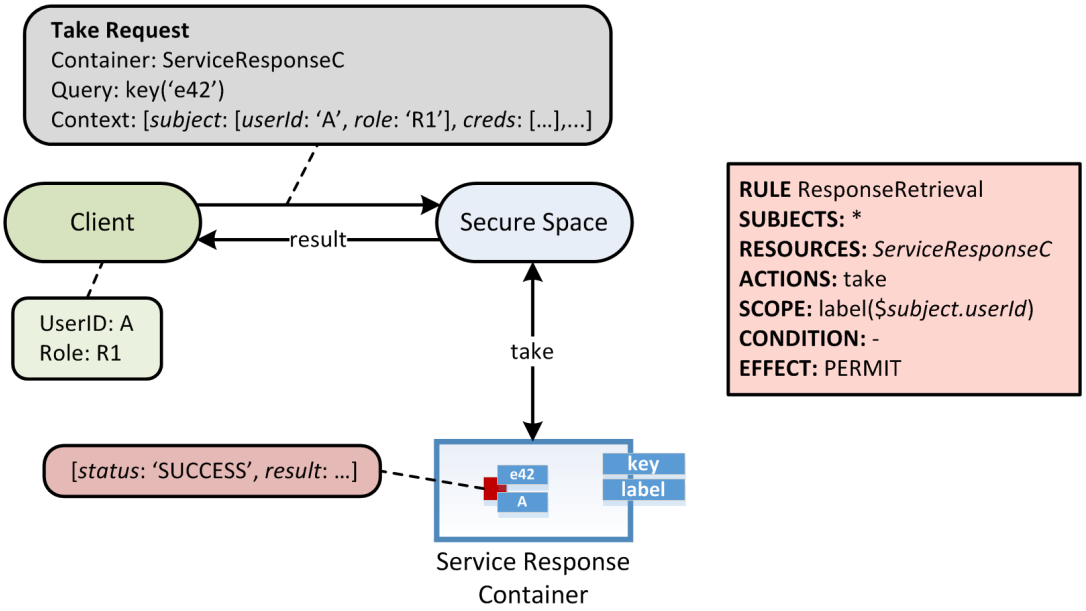

Figure 5.11: Response retrieval in the Secure Service Space (with default rule)

corresponding authorization rule on that space would have to include one or more security attributes of the service instead of the `localAccess` property, e.g., "[*userId*: 'Server123', *EA.DA.role*: 'R1']". A subject template must not contain solely delegated attributes, as they are not verified during the authentication phase and could be included in the request context by any invoker of the space. However, the combination of both principals in a subject template indicates that the authenticated user is trusted to act on behalf of the delegating user.

In some cases (e.g., for logging purposes), a service may also choose to perform a space operation using its own permissions without including the delegated attributes in the request context. Locally, this is implicitly allowed, whereas a regular direct access is performed on remote containers, using the associated security attributes and credentials of the service[6].

Response Retrieval

After execution, services write their responses into the service response container, which is implicitly authorized. The response either includes the result of the requested operation or an appropriate error message (e.g., if indirect container access has not been authorized). A service uses `label` and `key` coordinators to link the response to the authenticated user ID and the user-defined service request ID, respectively. Figure 5.11 depicts the

[6]The described approach assumes that a service shall either be permitted to access a container only directly or only using delegation on behalf of selected users, as subject templates from direct access rules also implicitly match any indirect remote access by the matched subjects. If both access types should be distinguished, an additional property (analogous to the `localAccess` field) is necessary.

retrieval of this response by the invoker, which uses a blocking or non-blocking take operation with the request ID as key. The associated rule ensures that only the principal that has issued the request may retrieve the response. Thus, even knowledge about the request ID does not enable attackers to access the response.

5.5 Implementation

To demonstrate the feasibility of the approach, the Secure Space architecture has been realized via the Java open source reference implementation of XVSM, called *MozartSpaces*[7]. MozartSpaces 2.0 [Bar10, Dön11] combines the XVSM specification [Cra10] with object-oriented design principles and a modular architecture. Compared to the original XVSM middleware architecture, there are some deviations for the sake of scalability and usability. Entries are represented as arbitrary objects, whose member variables correspond to properties in the XVSM model. System containers like the XVSM request container are replaced by components that actively forward requests to the next stage. The behavior of the coordination layer is largely outsourced to the container implementation, which is therefore the main interaction partner of the access manager. Furthermore, the CAPI has been extended with additional configuration parameters and overloaded method variants.

By adapting and extending this middleware, a prototypical version of the Secure Space has been created in cooperation with other members of the MozartSpaces development team. This implementation largely follows the security design described in the previous sections, although some of the less relevant features were omitted in the initial version. For instance, scope and condition queries are implicitly connected using disjunction and conjunction, respectively, while dynamic parameters are only supported for some selector values. This approach helps to keep the API for rule definition relatively simple. Notable extensions to MozartSpaces are described in the following.

Listing 5.1: Rule definition in MozartSpaces

```
1  ... // obtain cref via container creation or lookup
2
3  NamedValue roleA = new NamedValue("role", "RoleA");
4  NamedValue roleB = new NamedValue("role", "RoleB");
5  Subject tmpl1 = new Subject(Collections.singleton(roleA));
6  Subject tmpl2 = new Subject(Collections.singleton(roleB));
7  Set<Subject> subjTmpls = new HashSet<>();
8  subjTmpls.add(tmpl1);
9  subjTmpls.add(tmpl2);
10
11 Set<ContainerAction> actions =
       Collections.singleton(ContainerAction.TAKE);
12
```

[7] http://www.mozartspaces.org, accessed: 2020-04-09

```
13 LocalContainerReference localCref =
     new LocalContainerReference(cref.getId());
14 Set<LocalContainerReference> containers =
     Collections.singleton(localCref);
15
16 AuthorizationTarget target =
     new AuthorizationTarget(subjTmpls, actions, containers);
17
18 ContextAwareSelector<LabelSelector> selector =
     new ContextAwareLabelSelector(
       new Variable<String>("subject.userId", String.class, null),
       new Constant<Integer>(Selector.COUNT_ALL),
       new Constant<String>(LabelCoordinator.DEFAULT_NAME));
19 ScopeQuery scopeQuery =
     new ScopeQuery(Collections.singletonList(selector));
20 Scope scope = new Scope(Collections.singletonList(scopeQuery));
21
22 Condition condition = new Condition();
23
24 AuthorizationRule rule = new AuthorizationRule("Rule1", target,
     scope, condition, Effect.PERMIT);
```

Policy Specification. Following the object-oriented programming paradigm, rules are represented as complex `AuthorizationRule` objects, whose member variables represent the corresponding rule parts. A rule is created by instantiating its individual fields and passing them to the constructor. Listing 5.1 shows the definition of an example rule. At first, two subject templates with different role properties are created and added to a set representing the rule's `subjects` field (lines 3–9). Combined with singleton sets for the action (line 11) and the local container reference of the targeted container (lines 13–14), the authorization target is specified (line 16). The defined scope consists of a single scope query with a single selector (lines 18–20). To enable context-aware rules with dynamic parameters, a `ContextAwareSelector` interface has been introduced, which either wraps a regular XVSM selector or enables its dynamic computation based on the request context. The instantiation of the `ContextAwareLabelSelector` in the example (line 18) includes a dynamic parameter specified via the generic `Variable` class, which requires the property name within the context, the expected type, and an optional default value in its constructor. The count parameter and an additional coordinator name, which enables the distinction of multiple coordinator instances of the same type for one container, are specified as constants. Together with a unique name, an effect, and an empty condition (line 22), the rule can finally be created (line 24).

To simplify the specification of authorization rules, various helper functions could be implemented to enable the creation of common rule types (e.g., for role-based access on

a single container) without having to explicitly instantiate all rule components and their associated collections.

Listing 5.2: Rule activation in MozartSpaces

```
1  ... // initialize capi and policyC
2
3  Entry entry = new Entry(rule, KeyCoordinator.
       newCoordinationData(rule.getRuleId()));
4  List<Entry> entries = Collections.singletonList(entry);
5  TransactionReference tx = null;
6  RequestContext context = new RequestContext();
7  ... // set security attributes & credentials in context
8      // (only for remote access)
9
10 try{
11   capi.write(entries, policyC, RequestTimeout.TRY_ONCE, tx,
       IsolationLevel.REPEATABLE_READ, context);
12 } catch(MzsCoreException e) {
13   ... // error handling
14 }
```

Listing 5.2 shows how this rule can be included in the current authorization policy by writing it into the local policy container. Coordination data only needs to be specified for the `KeyCoordinator` (line 3), as the `VectorCoordinator` is not relevant for the currently supported combination algorithms and other coordinators of the policy container do not require additional information. The rule is activated using a non-blocking write operation with an implicit transaction (line 11). When writing to a remote policy container, the relevant security attributes and credentials have to be included in the request context. Compared to the XVSM specification, the method possesses an additional configuration parameter for the isolation level of transactions, which is, however, not relevant for this example.

Policy Enforcement. The access manager of the XVSM security architecture is realized with the `DefaultAccessManager` class, which provides a `checkPermissions` method for computing an authorization result for a specific operation. It retrieves rule entries with matching target from the policy container and passes them to a preconfigured combination algorithm (implementing the `ICombinationAlgorithm` interface), which evaluates the individual rules and returns a combined authorization result.

In order to integrate the access manager and thus facilitate the enforcement of the authorization policy, the coordination layer logic embedded within the `DefaultContainer` class and the coordinator implementations had to be adapted.

Authentication. The MozartSpaces development team has added an experimental authentication mechanism that supports SSO using the OpenAM framework, whereas user

management is performed via LDAP. Clients first authenticate at the identity provider, which returns an SSO token that has to be included in subsequent communication with the target space. For each request, the authentication aspect verifies the received SSO token, thus confirming the claimed security attributes. As the authorization mechanism does not depend on the used authentication strategy, alternative approaches (e.g., based on passwords or certificates) can be easily integrated into the middleware.

Integration of the Security Architecture. The relevant meta containers and aspects, as well as the access manager component are initiated when starting up the middleware instance. Authentication as well as invocation- and data-based authorization can be activated separately via the middleware configuration, whereas access control can also be configured at the container level via an additional parameter for the container creation method.

5.6 Benchmarks

Simple micro benchmarks have been designed and implemented to analyze the behavior of the MozartSpaces security extensions in typical usage scenarios, i.e., querying data from and sending requests to a server. They refine the initial performance analysis published in [CDJK12]. The main goals of this evaluation are the determination of the general overhead for access control as well as the scalability with policy complexity, amounts of data, and number of clients. Each examined scenario defines a specific space-based interaction, whose performance is measured in multiple benchmark runs that use increasingly complex authorization policies and varying benchmark-specific parameters. Naturally, performance depends on many factors, including entry sizes, used coordinators, operation parameters, and network configuration. Therefore, these micro benchmarks can only cover a few selected scenarios. Nonetheless, they provide relevant conclusions about the overhead of the examined access control mechanisms.

5.6.1 Test Setup

A framework has been implemented that executes the defined benchmarks with different configurations and measures their performance. In both examined cases, one XVSM core invokes functionality on another one that runs on the same host. Thus, no network latency occurs, but the overall communication overhead (including serialization) has to be considered. The focus lies on testing the authorization process. Therefore, authentication is only simulated by directly injecting a role attribute into the request context.

Like in previous runtime benchmarks [Dön11], mostly default values are applied for the configurations of the MozartSpaces cores. TCP socket communication is used in combination with the built-in Java serializer with a pool of 40 threads for sending and receiving messages. The XVSM core processors are directly executed within these threads and no separate thread pool is created.

A benchmark run for a specific scenario and configuration consists of five warmup rounds to reduce the effect of just-in-time (JIT) compilation by the Java Virtual Machine (JVM), followed by ten measured benchmark rounds, whose average is returned as a result. For each round, the cores are restarted and the involved containers are initialized. Then, the examined interaction is repeated for a specific number of iterations. Minor inconsistencies (e.g., lower execution time despite increased complexity) may be caused by effects of JIT compilation, garbage collection within the JVM, and concurrently running processes. Nevertheless, the standard deviation for all measured benchmark runs remained below 5%, which confirms the validity of the obtained results.

A standard PC with an Intel Core i5-2300 CPU (four cores at 2.8 GHz), 4 GB RAM, and Windows 7 Professional SP1 (64bit) was used as a test environment. The benchmarks were executed with the server VM of the Oracle JDK 1.8.0_172.

5.6.2 Data Query Benchmarks

In this scenario, a client repeatedly reads an arbitrary entry from a container on the server space using the `any` coordinator. The server core has to check that the client access is authorized. The performance for different access control configurations is compared. Additionally, the benchmarks examine their scalability with the total number of stored entries in the target container.

Figure 5.12 shows the results of these data query benchmarks, where 10,000 consecutive read operations are performed for each benchmark round. A configuration with deactivated security extensions serves as a baseline for the comparison. As arbitrary entry selection is used, the number of entries in the target container does not influence the performance. The basic overhead of the authorization framework can be determined via a configuration with a single authorization rule that permits every access via wildcards ("general authorization"), whereas the request authorization aspect is deactivated. The results show an execution time increase of 5–10% for all cases. When request authorization is activated in combination with an invocation-based rule that only allows read requests (using a `type` selector in its scope), the overhead approximately doubles, as two authorization checks (on the request container and the target container) are required.

For the next policy configuration ("simple rule"), access to the target container is controlled by a rule that allows the client role to access entries in the container with a specific label (using a `label` selector in its scope), which is set for half of the available entries. For low entry numbers, there is no significant difference to the previous test. However, with 1,000 entries, the total overhead increases to 22% (compared to 11% for 10 entries). This can be explained by the fact that a decision mapping has to be created for all target container entries. The execution time for high entry numbers increases significantly if the scope includes certain selectors that need to be evaluated individually for each entry. In the "complex rule" setting, the scope of the previous rule is extended to also check that a specific entry property is below a certain threshold (using a `query` selector), which reduces the number of allowed entries to a fourth of all available ones. Due to the costly evaluation of the scope by the access manager, the overhead increases to 63% for 1,000 entries. The addition of a condition with a dynamic parameter to the

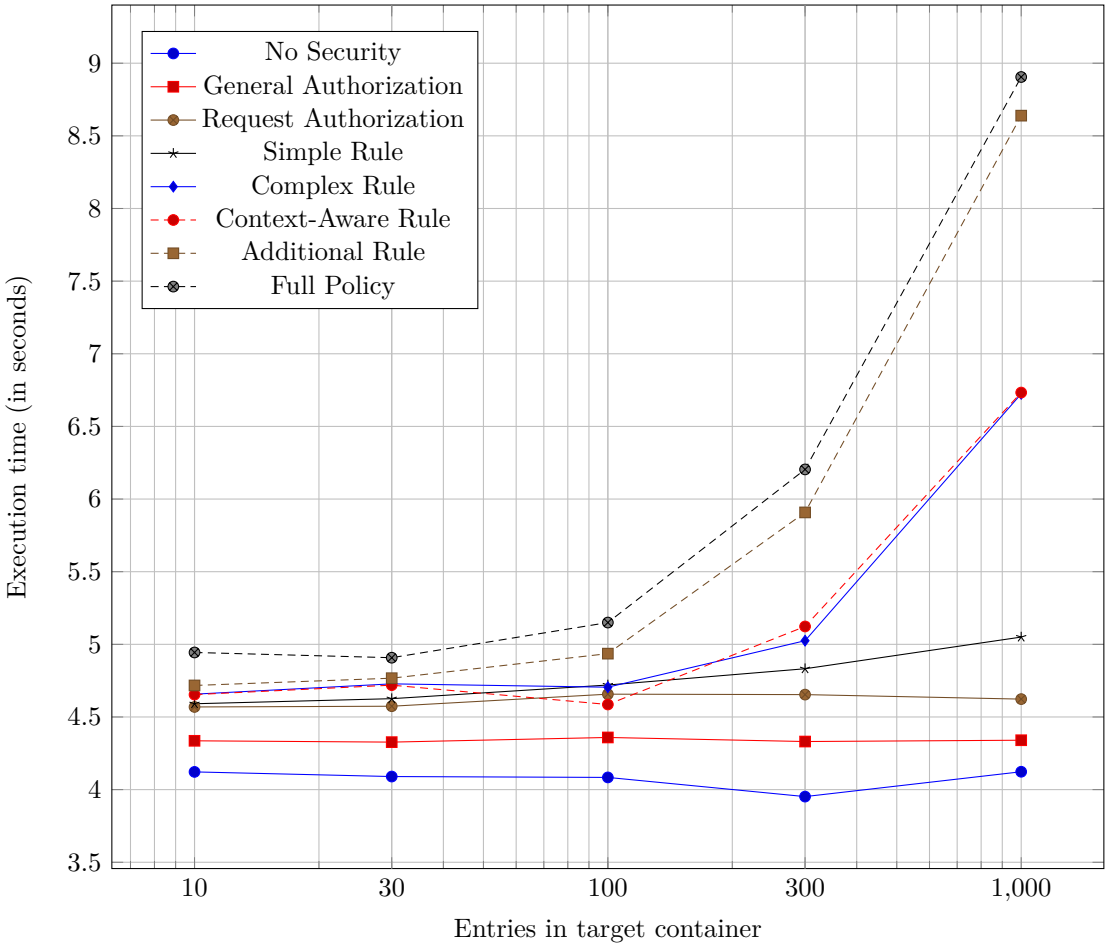

Figure 5.12: Data query benchmarks with 10,000 iterations

rule ("context-aware rule"), which checks for an entry with a client-defined label in an otherwise empty context container, does not seem to affect performance.

When multiple rules apply for an operation, all of them have to be evaluated. Therefore, when a second, identical rule is added, also the time required for authorization increases significantly to 110% in the worst case. Rules that are not applicable (i.e., targeting different subjects, containers, or actions) have less impact. In the "full policy" configuration, 100 additional rules are added, which simulates a rather complex authorization policy, but the execution times only go up by 3–5% compared to the previous configuration.

These benchmarks show a reasonable overhead for authorization of query operations in most cases. Scalability with the number of entries depends on the complexity of the authorization policy, mainly on the used selectors within `scope` fields. For rules with simple scopes and conditions (e.g., using a `label` selector), the developed extensions provide good scalability. However, if more than 100 entries are expected in a single con-

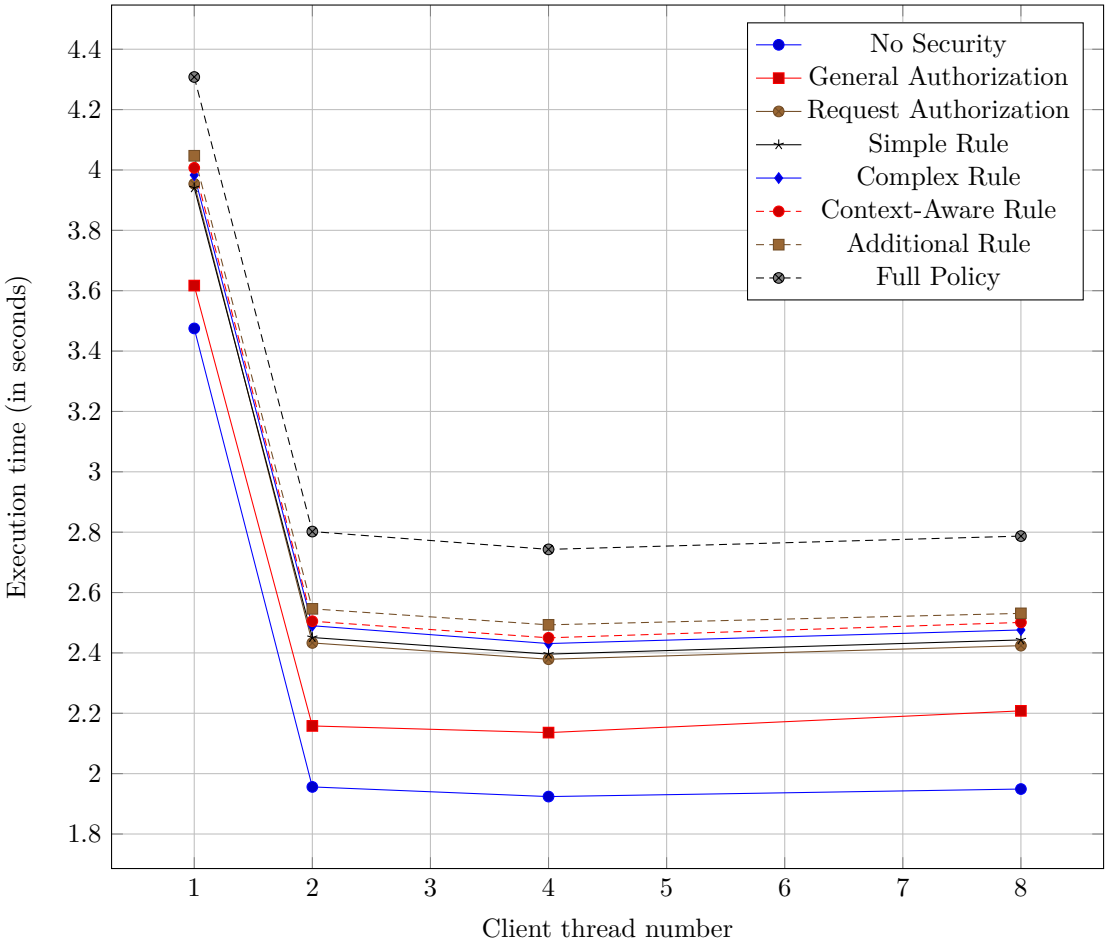

Figure 5.13: Concurrent request execution benchmarks with 10,000 iterations

tainer, which does not apply to a large portion of coordination scenarios, low performance coordinators should be avoided within the scope.

5.6.3 Concurrent Request Benchmarks

This scenario includes client threads that concurrently write request entries to a target container at the server space. Each written entry is immediately taken by the local server process using blocking take operations, which resembles the service invocation approach of the Secure Service Space. As before, the performance of the client authorization process is analyzed using different policy configurations. To evaluate the concurrency behavior of the framework, the benchmarks are repeated for different numbers of client threads.

Figure 5.13 shows the results of these tests. They depict the average execution time for 10,000 total write operations, which are split evenly among the corresponding

number of client threads. The results indicate that the authorization mechanism does not substantially affect scaling with the number of connected clients. In all examined configurations, a significant speed-up is measured when operations can be processed in parallel. The best performance is reached with four threads, which corresponds to the number of available CPU cores for the test environment, although the difference to the benchmarks with two and eight threads is minimal.

The basic overhead of authorization for write operations reaches up to 13% for the general authorization setup and up to 24% when type-based authorization on the request container is activated. In the single-threaded setting, these values are lower (4% and 14%, respectively), which indicates that the access control mechanisms do not profit as much from the parallelization as other components of the MozartSpaces runtime. As the server process continuously clears the target container, scope evaluation is relatively fast and the complexity of applicable rules is less significant for the total performance. The simple rule configuration allows write access for a specific entry type (using a scope with a `type` selector), whereas the complex rule additionally includes a `query` selector that checks whether an operation name property within the entry equals a specific value. For the context-aware rule, the same condition as for the data query benchmarks is applied. Even when using two complex, context-aware rules, the maximum overhead is 30%, i.e., only slightly higher than for the request authorization configuration. An extended policy size with 100 additional rules further increases the execution time by 6–10%.

The benchmarks show that write operations generally have a slightly higher relative overhead than query operations, which can be mostly explained by the fact that their overall processing time is shorter, thus the authorization time covers a bigger share. The performance of the authorization extension is still sufficient and it supports concurrent handling of XVSM requests that scales with the number of available CPU cores. For the authorization of service requests and similar use cases, rather complex policies could be defined without degrading performance. However, as scope checks always involve the whole container, also write operations may be slowed down by complex authorization policies when writing into containers with many entries.

5.7 Critical Reflection

The presented approach supports expressive yet comprehensible policies for open distributed scenarios while following state-of-the-art security principles. The access control model is highly versatile, as it supports simple ACLs as well as complex content- and context-based constraints. The integration of middleware concepts into the policy language enables developers and administrators to coordinate application components and manage access control using similar mechanisms.

However, there also exist some drawbacks for the described access control extensions. One main issue is the performance overhead, which largely depends on the complexity of the applied authorization policy and the number of entries in accessed containers. This tradeoff between expressiveness and scalability must be considered when managing policies for performance-critical applications. For instance, the usage of general decision

objects significantly speeds up access decisions when no `scope` fields are defined. All in all, the access control overhead should be less severe than other limiting factors, like network latency.

Another potential problem is that the semantics of coordinators (especially custom ones) may not always be fully clear, which can lead to misconfigured policies. As aspects are able to manipulate all parameters of an XVSM request (including the context), they can potentially bypass the access control mechanism. Therefore, permissions for adding aspects to a space should only be given to fully trusted subjects.

While the Secure Service Space supports indirect access, chained delegation (e.g., "*A* **for** *B* **for** *C*") has not been considered. Furthermore, the provided service architecture is merely defined as a coordination pattern, which means that its functionality and security rely on the correct implementation of the service framework and all installed services.

Other open issues are related to the prototypical implementation within MozartSpaces. Future versions should provide a complete feature set and incorporate a more usable way to specify authorization rules, possibly via an improved API or a graphical management tool. Additionally, the resilience of the current middleware runtime against various attacks needs to be reexamined, which includes the support for encrypted communication channels and the establishment of a secure middleware kernel that can act as a TCB.

XVSM facilitates coordination in a flexible way, but at a rather low abstraction level that directly follows the SBC paradigm. A more high-level, model-based design approach is pursued by the Peer Model, which combines the container concept of XVSM with the specification of complex workflows. In the next chapter, an advanced access control model for the Peer Model is presented that is based on the concepts of the Secure Service Space, but mitigates most of its drawbacks.

CHAPTER 6

The Secure Peer Space

The Peer Model [KCJ+13] is a high-level coordination model for data-driven workflows in complex distributed systems. It combines concepts of SBC, flow-based coordination, and message-oriented approaches like the Actor Model. In order to enable a decoupled communication style, space containers form input and output stages for distributed application components that encapsulate coordination logic in a structured manner. This resembles the space-based service invocation pattern described in Section 5.4, but in contrast to the Secure Service Space, the interactions with the containers are specified separately from the service code in a declarative way. Thus, separation of concerns between coordination logic and application logic is achieved.

The Peer Model may be used merely as a design language for concurrent and distributed flows (similar to UML), but its main application field is model-driven development, where designed models form runnable specifications that can be executed and dynamically adapted via a corresponding middleware runtime, i.e., the Peer Space. This approach bridges the gap between design and implementation in the domain of distributed coordination. The long-term vision for the Peer Model is the provisioning of a complete toolchain, including a graphical modeler and verification tools [KCH14]. The Peer Space shall be implemented on different platforms (including embedded devices) in an interoperable way, so that it can be used for P2P coordination in heterogeneous environments.

Like XVSM, this middleware must be secured by means of suitable access control mechanisms. The model-driven approach of the Peer Model can be enriched with declarative authorization rules, thus forming a *Secure Peer Model* that allows developers to model both coordination and access control constraints. As all communication is performed via space containers, the previously defined security architecture for the Secure (Service) Space can be adapted in order to realize the Secure Peer Space [CJK15]. Section 6.1 provides an overview of the Peer Model and the Peer Space middleware. Section 6.2 specifies a suitable access control model, while Section 6.3 describes the corresponding architecture of the Secure Peer Space. Section 6.4 outlines how this

approach can be adapted for resource-constrained wireless communication devices, as required by the LOPONODE use case. The prototypical implementations of the Secure Peer Space are described in Section 6.5, while Section 6.6 covers initial benchmark results. Finally, Section 6.7 provides a critical reflection of the approach.

6.1 Peer Model Overview

The Peer Model is based on the notion of *peers*, which represent autonomous components in a potentially distributed application. Communication between peers and the management of internal peer states are realized via an SBC approach, where each peer hosts a set of predefined containers. The coordination logic of a peer is determined by its *wirings*, which are triggered by a specific combination of entries in the peer's containers. They can consume the triggering entries, invoke application logic in the form of services, and emit the resulting entries to local or remote containers. These entries may then trigger further wirings in the targeted peer and so forth. Eventually, a single entry can lead to the execution of a complex *flow* involving wirings in several distributed peers.

In the following sections, the relevant modeling concepts and their realization via the Peer Space middleware architecture are described. As the Peer Model is still an evolving concept, different variants with varying features have been described [Küh16, Küh17, KRE18]. In this thesis, only the basic model is covered [KCJ+13, KCJN14, KCH14], with a few extensions from later versions, mainly related to assignments and variables [Küh16]. To ensure consistency with ongoing specification efforts, some terminology has been adapted and incompletely specified parts have been expanded (e.g., the meta model [CJK15]).

6.1.1 Basic Modeling Concepts

In the Peer Model, applications are modeled in a declarative way by specifying peers and wirings using a DSL or a corresponding graphical representation. In this work, a graphical Peer Model notation is used, as it provides a clearer overview of the modeled behavior. At run time, the current state of an application is represented by the contents of all distributed peer containers, which are continually updated according to the modeled wirings. Additionally, external processes can trigger flows by injecting entries into containers. Figure 6.1 shows the graphical representation of a simple peer with a single wiring. In the following, the meaning of the different elements is explained.

Peers and Containers. Each peer depicts a logical component with a unique name that consists of an input stage in the form of a *Peer-In-Container (PIC)*, internal behavior specified by a set of wirings, and an output stage in the form of a *Peer-Out-Container (POC)*. Optionally, additional containers may be used, e.g., to store internal data. The containers provide XVSM-like functionality, where entries can be written, read, or taken with support for continuous queries and transactional access. Peers may also contain (potentially nested) sub-peers. After being deployed

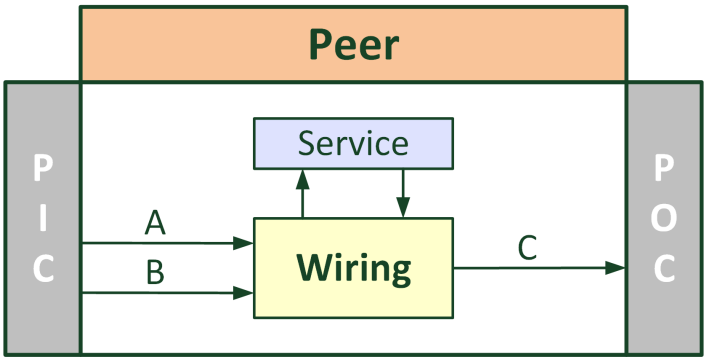

Figure 6.1: Graphical notation for a simple peer

to a specific Peer Space, peers can be addressed via their URIs, using the structure "*PeerSpaceURL*/*PeerName*/*SubPeerName*/*SubSubPeerName*/...". If no Peer Space is specified, the local one is implicitly selected. Containers can be addressed by appending their label to the peer address, e.g., "*Peer1*/*POC*". By default, the PIC is used when a peer address is given[1].

Entries. As in XVSM, entries consist of (possibly nested) properties. Each entry contains a mandatory `TYPE` property that specifies its coordination type as well as an internal entry ID (`EID`) that can be used as a reference[2]. A set of predefined system coordination properties with fixed semantics affect its handling by the Peer Space runtime:

- The `TTL` (time-to-live) property specifies when the entry expires, i.e., when it is automatically removed from its container.
- Delayed activation can be achieved via the `TTS` (time-to-start) property, which specifies when an entry becomes visible within a container.
- The `DEST` (destination) property includes the address of the entry's intended target container.
- Different flows (i.e., workflow instances) can be distinguished via the `FID` (flow ID) property.

When creating an entry, users can additionally include custom coordination properties, which are referenceable within the modeled coordination logic, as well as relevant application data for services.

[1] To avoid conflicts, this addressing scheme requires that namespaces for containers and peers are disjoint, i.e., there must not be a sub-peer called "PIC" or "POC".

[2] The Peer Space runtime must ensure that distinct entries have separate EIDs.

Query Semantics. Instead of supporting the full expressiveness of XVSM queries based on the coordinator concept, the basic variant of the Peer Model uses simplified *Peer Model Queries (PMQ)*, which combine type-based access with a flexible count mechanism and entry-specific predicates. A PMQ consists of three parts according to the following structure:

$$type\ [count]\ [\![selector]\!]$$

The mandatory *type* specification defines a string that is matched with the type property of entries within the container. The *count* value determines how many entries have to be selected. It is specified as a range in the form "$min; max$", whereas a single value can be used as a shortcut if $min = max$. The special value `ALL` stands for the number of all matching entries, which may also be zero. If the minimum value is zero or the maximum value is unbounded, a single value can be combined with a relational operator to indicate the range. If no count is specified, the default value of one is used. The optional *selector* parameter maps to a subset of the features of the XVSM `query` selector, which is restricted to predicates that can be evaluated separately for each entry. This includes comparison operations for specific entry properties combined with logical operators. In order to satisfy a PMQ, there must be at least as many entries of the given type with matching selector (if specified) as defined by the minimum count. In this case, matching entries are returned in a non-deterministic order until the maximum count is reached or no more entries are left. Only valid entries are considered for a container query, i.e., entries that are currently active according to their `TTS` and `TTL` properties.

A formal specification of the PMQ syntax is given in the appendix (Section A.3.2). As an example, the PMQ "Packet $[\geq 2]$ $[\![weight \geq 20 \wedge weight \leq 100]\!]$" requires at least two entries with type `Packet` and a `weight` property whose value lies between 20 and 100.

Wirings. Wirings connect containers using a conditional link mechanism that resembles well-known concepts like CPN transitions, Reo channels, and SALSA tasks. Thus, they form the active parts of a peer that realize its coordination logic. A wiring consists of one or more *guard links* (or *guards*), an optional service, and zero or more *action links* (or *actions*). Each link belongs to a specific link type and contains a PMQ that acts as a condition. `Move` links take matching entries from their source container and write them into their target container, whereas `copy` links use a read operation instead. The target of a guard and the source of an action is an *entry collection*, which represents a temporary container implicitly created for each *wiring instance* (i.e., for each execution of a wiring). Valid guard sources and action targets must be located in the enclosing peer or in one of its direct sub-peers. Additional link types that only involve a single container are possible (e.g., to delete entries without forwarding them or to test if an entry is not present in a container), but they are all based on the standard container operations of read, take, and write.

A wiring is triggered when all of its guards are satisfied, i.e., the respective read and take operations are successful. Using the retrieved entries from the entry collection

as input, the service is subsequently invoked, which performs its application logic and returns a result entry list that replaces the content of the entry collection. Then, the actions are performed on this updated entry collection. In the example from Figure 6.1, the two guard links consume one entry of type `A` and one entry of type `B` from the PIC using the default link type of `move` and simple PMQs with default count and no selector. The sole action expects that the specified service use this input to create an entry of type `C`, which is subsequently transferred to the POC.

Conceptually, multiple wiring instances may be executed concurrently, but consistency of the affected containers has to be preserved via transactional access mechanisms. Therefore, the atomic execution of each wiring instance is assumed, i.e., any direct container change by a link becomes only visible after the wiring completes successfully. Furthermore, guards cannot access entries that have already been locked by a concurrent wiring instance.

Service Definitions. Within a wiring specification, services are integrated via a distinct service name that references application-specific code in the form of a service function. This function may access all properties of the input entries, modify them, and create new entries. It may also perform arbitrary computations and call external modules (e.g., a database or a GUI). However, services should terminate quickly and must not block indefinitely. Instead, asynchronous calls shall be used when invoking external components, where the response is injected back into the space and treated by another wiring. A service also cannot directly access containers of the Peer Space. This ensures that the service only consumes and emits entries that were planned in the designed coordination logic via the guard and action links of the associated wiring. Compared to service invocation in the Secure Service Space, this approach provides a more controlled execution environment.

Flow Correlation. In order to allow for the concurrent execution of multiple workflow instances, a flow correlation mechanism based on the `FID` property is applied. Entries with different flow IDs cannot be used together to trigger a wiring. For each wiring execution, guard links must ensure that all selected entries either belong to the same flow or are not part of any flow (i.e., no `FID` property is set). When creating a new entry, it can be assigned an existing flow ID (e.g., of an input entry) or a new flow ID that is automatically generated by the middleware. This mechanism can be used, for instance, to correlate a previously issued request entry with the received response entry.

Explicit Addressing. Remote communication and the dynamic selection of recipients is realized via the `DEST` property. Whenever an entry with set `DEST` property is handled by an action targeted at the POC, it is transferred to the specified destination container instead. Conceptually, this behavior can be bootstrapped by a set of implicit wirings that move such entries across peer and space boundaries until the destination container is reached. This approach enables flexible communication like in the Actor model, as

independent actors (i.e., peers) can communicate asynchronously by writing messages to each other's mailboxes (i.e., containers).

Assignments and Variables. In order to further decouple coordination and application tasks, entry properties related to the coordination logic may be directly specified by guards and actions. Therefore, links optionally contain one or more *assignment* statements, which enable the definition of specific property values (e.g., a common destination) for all entries that are transported on a link, overwriting any previously defined value if present. To enrich the semantics of this mechanism, also *local variables* are supported, which allow for the propagation of values between different links within a single wiring execution. They start with a "$" sign to distinguish them from entry property identifiers. In addition, also *system variables* can be referenced, which are automatically set by the runtime and start with "$$". Examples include the current time (`$$TIME`) and the address of the current peer (`$$THIS_PEER`).

Assignments are expressed in the form "*id* = *expr*", where *id* denotes a property or variable identifier and *expr* an expression that can be resolved to a value at the evaluation time of the link. Such an expression may contain concrete values, system variables, previously defined local variables, and references to specific entry properties. Combination via predefined functions and operators is supported, e.g., for list size, arithmetic operations, or string concatenation. A formal definition of the syntax can be found in Section A.3.3 of the appendix.

Within this thesis, local variables are only assigned by guards and used by actions. This simplifies wiring execution by preventing dependencies between links of the same phase. As an additional restriction, entry property assignments are only used for actions. Within a guard, local variables may be assigned to the value of a referenced entry property[3]. Local and system variables can subsequently be used to dynamically define entry property values within actions. In a similar way, also values of other properties from the same entry may be involved.

Exceptions. During the execution of a modeled application, several types of *exceptions* can occur, which may be caused by the Peer Space runtime or by services. System-specific exceptions include the expiration of an entry due to its TTL and communication failures with remote spaces. The exception mechanism is bootstrapped with already existing features. Exceptions are created as regular entries with a corresponding exception type that are automatically written to a container of the originating peer. Like regular coordination logic, their handling can be modeled by the user by means of wirings.

Coordination Example. Figure 6.2 illustrates several of the previously described advanced Peer Model concepts by means of a typical coordination example related to the generation and distribution of tasks. A server peer accepts incoming `Request` entries

[3]When the corresponding link allows more than one entry, a specific property may have different values for each entry. As this would cause ambiguity for the variable assignment, local variables should only be defined on links with a count of one.

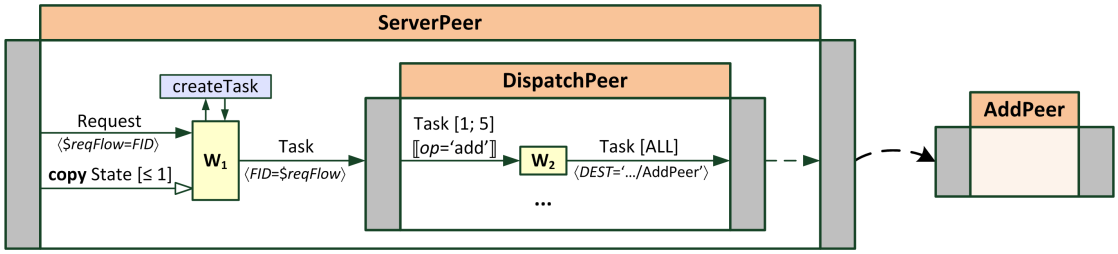

Figure 6.2: Peer Model example for state-dependent task generation and dispatching

and transforms them into corresponding `Task` entries according to the service logic of its wiring W_1. If available, a `State` entry is also passed to this service (as a copy), which may influence certain task parameters. Depending on the remaining server logic, this state may refer to the whole server or just the flow related to the current request. The flow correlation mechanism ensures that the right entry is retrieved in the latter case. In order to continue the flow, assignments are used to transfer the flow ID from the `Request` to the `Task` via the local variable `reqFlow`.

The result of W_1 is then written in the PIC of a sub-peer that dispatches tasks to different target peers according to their operation type. As an example, the figure includes the service-less wiring W_2, which retrieves up to five "adding" tasks (according to the entries' `op` property) and forwards all of them to a predefined peer via the destination mechanism. The assignment on the action sets the `DEST` property of all passed entries to the address of the target peer, which may be located on a remote space.

For the graphical notation, a simplified variant of the full graphical Peer Model representation [Küh16, Küh17] is used[4]. PMQs (as well as link types other than `move`) are generally depicted above their corresponding links, while assignments are written below them using angle brackets. The implicit wirings induced by the destination mechanism are indicated via dashed arrows.

6.1.2 Meta Model

To enable the dynamic adaptation of coordination logic, a meta model approach is applied. However, instead of using explicit meta model operations as specified in [KCJ+13], the installation of peers and wirings is bootstrapped via specification entries in special meta containers. Therefore, a *Wiring Specification Container (WSC)* and a *Peer Specification Container (PSC)* are added to each peer. Wirings are installed by writing a corresponding *wiring specification entry* into the WSC of the enclosing peer, while sub-peers can be added in a similar way via *peer specification entries* in the PSC. For the definition of peers and wirings at the top level in a Peer Space, an implicit *Runtime Peer (RTP)* is used that hosts all user-defined peers as sub-peers. Its address corresponds to the URL of the hosting Peer Space, while within the local runtime it is simply represented as "/".

[4]Features that are not necessary for the described access control mechanism and coordination examples are omitted. This includes explicit link order and various configuration properties.

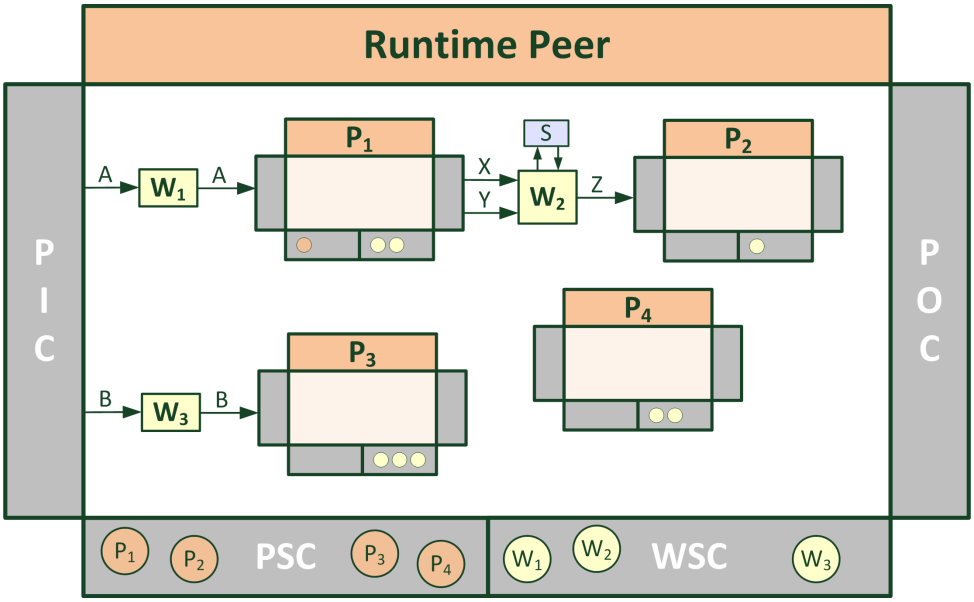

Figure 6.3: Runtime Peer with sample model and meta containers

Figure 6.3 shows how a model with four peers and three wirings is represented in the meta model via specification entries in the PSC and WSC of the Runtime Peer. Due to the configured wirings, incoming entries with type `A` are forwarded to peer P_1, while entries with type `B` are processed by P_3. Peer P_2 is triggered by certain outputs from P_1, whereas the invocation of P_4 is not modeled explicitly, i.e., it can only be invoked directly by using the `DEST` property. The internal coordination logic of these peers, which is not shown in the figure, is itself specified via entries in their respective specification containers.

Specification entries have a special type (`Wiring` or `Peer`) and store their behavior in structured specification properties (see Section A.3.4). Whenever the content of a meta container changes, the Peer Space runtime dynamically updates the model. The behavior of a wiring is fully defined via its distinct name, lists of guard and action link specifications (including source or target container name, a link type, a PMQ, and possible assignments), and an optional service reference. In contrast, a peer specification entry only defines the peer name. The behavior of the peer itself is defined recursively by adding entries to its own PSC and WSC, which are automatically created by the runtime when the peer is instantiated. Both specification entry types also include additional configuration parameters that affect the runtime behavior but are not further detailed here.

Other than that, specification entries are treated like regular entries. They can be accessed by local wirings because PSC and WSC are valid source and target parameters for links. Dynamic adaptation of the coordination logic is possible as wirings may create, modify, or delete these meta entries when their guard conditions are fulfilled. Limited lifetime and deferred activation can be configured via `TTL` and `TTS` properties,

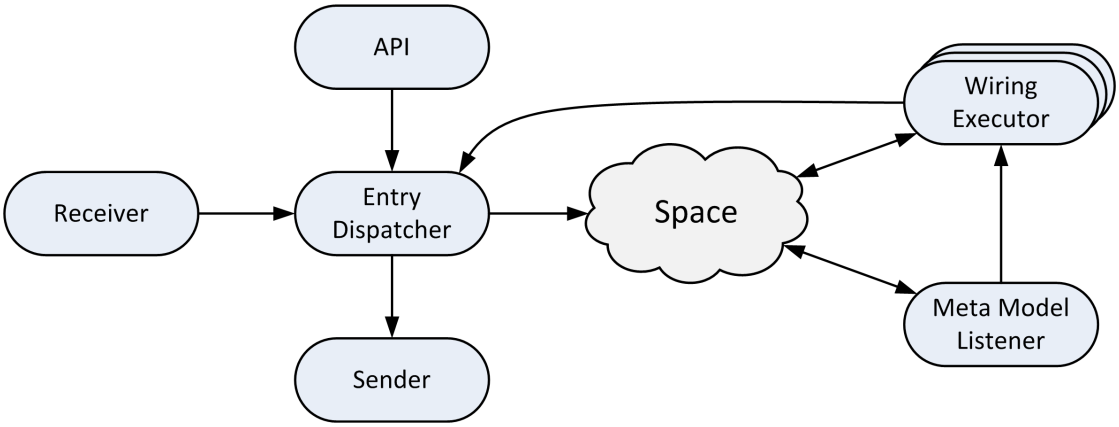

Figure 6.4: Peer Space runtime overview

respectively, while remote installation of coordination logic is enabled via the destination mechanism.

Dynamic wirings are not only used for updating the coordination logic at run time. They are also necessary for modeling dynamic query operations whose parameters are not known at design time, e.g., because they depend on entry data or service logic. Such a query is performed by creating a corresponding wiring specification entry (e.g., within a service) and sending it to the WSC of the targeted peer using the `DEST` property or a direct action link (for local access within the own peer or its sub-peers). This dynamic wiring then retrieves the requested entries via its guards and sends them back to the PIC of the requestor peer, where other (predefined) wirings can process the result. Instead of a continuous subscription, often only a one-off query is required. Therefore, wiring specification entries may include an optional repeat count property that specifies how often they can trigger. If this value is set to one, the wiring is automatically removed after it has returned the query result.

6.1.3 Peer Space Middleware Architecture

Similar to XVSM, a Peer Space instance is embedded into a hosting application that can use an API to interact with the local Peer Space as well as remote cores. The runtime architecture, which is depicted in Figure 6.4 in a simplified form, consists of a space middleware that manages peer containers and additional components for realizing wiring execution, dynamic model changes, and remote communication.

The Peer Space features a simple API with only one major method: `addEntry`(*Entry*). This method is used to configure the desired coordination logic (via peer and wiring specification entries) and trigger its wirings by means of injected entries. The target container is determined by the entry's `DEST` property. For local containers, the *entry dispatcher* uses a write operation on the embedded space, whereas the sender component forwards entries with remote addresses to the receiver component of the targeted Peer Space. Additionally, the Peer Space supports methods for starting and stopping wiring

execution on the local core, which may be used to prevent inconsistent behavior during major model changes.

The space middleware hosts the containers of the RTP and all of its direct or indirect sub-peers. It has to support transactional access, PMQ-based query operation, lifecycle management for entries according to their `TTS` and `TTL` properties, and asynchronous notifications about changes in the space. XVSM with certain eventing extensions [CKSW17] is a suitable candidate, but alternative SBC middleware systems are also possible.

The *meta model listener* gets notified by the space about changes in any PSC or WSC. For new peers, it reacts by creating the required containers in the space, whereas deleted peer specification entries lead to the destruction of all associated containers (including those of sub-peers). For each active wiring, a *wiring executor* thread is started, which is stopped again when the wiring specification entry (or its WSC) is removed. In order to enable the parallelization of tasks, also multiple threads per wiring may be configured.

These wiring executors run concurrently in a non-deterministic order. They monitor the space according to their guard links by means of multiple query operations. When triggered, they execute the associated service (if defined) and evaluate their actions. When the `DEST` property of an emitted entry is not set, the wiring executor writes it directly into the action's target container, otherwise the entry dispatcher is invoked, which operates asynchronously in a separate thread. To ensure consistency, each wiring is executed atomically by using a transaction for its container operations. Either all required links are successful or none. When one or more guards are not satisfied, the wiring instance must immediately unlock any entries selected by its other guards. As services and actions should never block, entry locks are only held for a short time.

Additional functionality can be bootstrapped via predefined *system peers*. Connector peers act as proxies for external components like databases. As direct container queries are not possible via the Peer Space API, such a mechanism is also used for returning the results of a flow to the host application. Entries sent to a special connector peer trigger a wiring whose service invokes a configured callback function of the application. A meta model peer may simplify the modeling process by transforming a single specification entry (e.g., containing coordination logic in DSL format) into corresponding peer and wiring specification entries for distributed meta containers. Further system peers may provide general-purpose coordination functions, like distributed transactions or a peer lookup.

6.2 Access Control for the Peer Model

The core principle of the Peer Model's access control model is the authorization of operations on peer containers. This container-centric authorization approach is realized by adapting the previously described XVSM access control model to the needs of the Peer Model. Thus, expressive authorization rules enable fine-grained access control for each peer container. In contrast to the Secure Service Space, where services are implicitly trusted, any authorized user may inject coordination logic into a Peer Space in the form

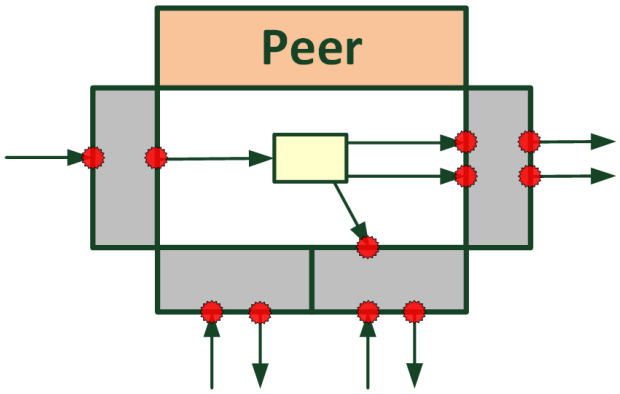

Figure 6.5: Required authorization checks for interactions in the Peer Model

of wirings. Therefore, not only external write access from remote Peer Spaces, but also local query and write operations by wirings must be controlled. Figure 6.5 visualizes the required authorization checks for an example peer. It shows that any external access to the peer's PIC, POC, and meta containers (by means of wirings or entry injection via the destination mechanism) must be authorized. In order to control what dynamic wirings are allowed to do, authorization checks also need to be performed for internal container access by wirings within the peer. Without these internal checks, fine-grained access control on coordination logic injected by potentially not fully trusted subjects would not be possible. Instead, either all injected wirings would have full access, or external changes to the WSC would need to be denied completely.

As for XVSM, it is assumed that incoming entries from remote sources can be authenticated with the help of a trusted identity provider and that secure communication channels are established. Following the ABAC approach, user identities are represented via a set of authenticated security attributes. However, the emphasis on complex flows involving multiple stakeholders, multitenancy support, and dynamic adaptation of coordination logic requires a more sophisticated notion of subjects than in the Secure Service Space. Therefore, a novel mechanism based on the concepts of ownership, chained delegation, and trust has been developed.

6.2.1 Subject Concept with Delegation Support

Access control in XVSM targets CAPI requests that have been enriched with corresponding user credentials by the invoking client. In the Peer Model, however, most container operations are not directly triggered by external processes, but by internal wirings. As these wirings operate within a controlled runtime environment, they should not be required to handle credentials of their associated principals themselves. Instead, authentication is transparently handled by the middleware, which greatly reduces complexity for developers. However, the subject of each operation still needs to be defined in order to enable authorization. Therefore, the secure version of the Peer Model introduces the concept of ownership for entries, peers, and wirings.

The runtime associates each incoming entry with a responsible subject according to its source. The authenticated security attributes of this *entry owner* are stored within the entry's subject property, which is an additional system property for access control purposes. When entries are written into a container, this information is used to check the authorization of the corresponding write operation. As peers and wirings are also specified via entries, their respective owners are defined in the same way. A *peer owner* (i.e., the entry owner of the corresponding peer specification entry) is responsible for configuring the peer's authorization policy, which is described in Section 6.3. Whenever a wiring accesses a container via its guard or action links, its associated *wiring owner* (i.e., the entry owner of the corresponding wiring specification entry) is relevant for the authorization process.

For read and take operations by guards, the wiring owner is used as the responsible subject. A wiring can only be triggered if its owner has the necessary permissions. For write operations via actions, wirings can choose between *direct* and *indirect access*, similar to services in the Secure Service Space architecture. This selection can be configured separately for each emitted entry. It determines the responsible subject for the authorization of the write operation, which is also set as the respective entry owner. For direct access, the wiring owner is used as the subject, while for indirect access, the wiring owner acts on behalf of another subject, which is selected among the owners of all input entries. In this case, a compound subject containing both the wiring owner and the respective entry owner is required. This mechanism enables delegation, but also ensures that wirings cannot impersonate arbitrary users. The identity of the wiring owner is always included in the subject of an emitted entry, even when the entry itself is not changed and simply forwarded by the wiring.

Identity Representation. Similar to XVSM, each principal is represented via a set of properties provided by the authentication mechanism. Federation is considered by means of a `domain` property, which specifies a URI that relates to the user's organization or a corresponding community. In combination with the `userId` field, each user can thus be uniquely identified. Depending on the identity provider, additional properties may provide role information or other relevant security attributes. In order to support the coexistence of multiple authentication mechanisms with different trust levels at a single core, relevant information about the authentication process, like the URI of the identity provider or the used authentication method, are included in a nested `authContext` property.

Delegation Chains. A flow may involve several wirings with different owners. If each of them applies indirect access mode, chained delegation is enabled. A *delegation chain* includes all principals involved in the creation of a specific entry. It is represented by an ordered list of identities (e.g., "⟨[*userId*: 'User1', ...], [*userId*: 'User2', ...], [*userId*: 'User3', ...]⟩"). Its first element depicts the *originator* of a request, while the last element corresponds to the *invoker* of the latest write operation. For increased readability, a simplified notation is used that follows the representation of delegation chains in related

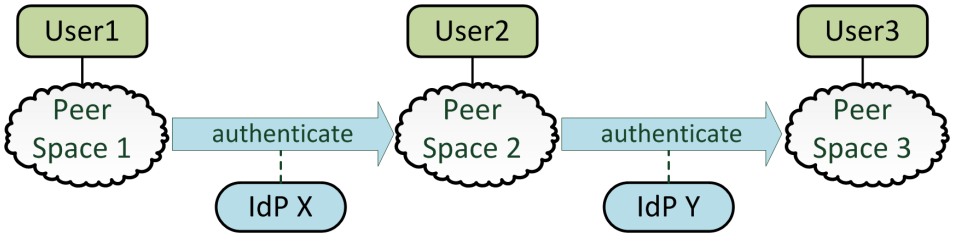

Figure 6.6: Chained authentication with different identity providers

work (e.g., [LABW92]). Principals are represented by their unique IDs and listed in reverse order separated by the keyword "**for**". The aforementioned delegation chain can thus be abbreviated as "*User3* **for** *User2* **for** *User1*".

For instance, user A may inject a request entry into a peer to start a flow. A wiring owned by user B fetches this entry and emits another entry with the compound subject "B **for** A". This entry may then be used by a second wiring (with owner C) to create an entry with subject "C **for** B **for** A" and so forth. A wiring owner can also be a compound subject if the corresponding wiring specification entry was written using indirect access. When such a wiring applies indirect access mode itself, the delegation chains of both involved subjects are simply merged, i.e., a wiring owner "D **for** C" may act on behalf of an entry owner "B **for** A" by emitting an entry with owner "D **for** C **for** B **for** A".

Authentication Chains. Following the P2P paradigm, authentication is performed in a decentralized way, i.e., each Peer Space is responsible for authenticating incoming entries from remote sources. For this, runtimes rely on a set of trusted identity providers, which may be different for each core. When flows span multiple Peer Spaces (using the destination mechanism), also the runtimes themselves have to be authenticated when transmitting entries on behalf of the respective entry owners. Therefore, each Peer Space is assigned a corresponding identity in the form of a *runtime user*. There is a difference if all principals within a delegation chain have actually been verified at the local core or if another core merely claims that it has authenticated these users. In the second case, the runtime can only authenticate the runtime user of the remote core and has to trust it to tell the truth about the delegated attributes.

If multiple Peer Spaces are involved in a flow, a chain of trust is required as each Peer Space only authenticates cores from which it directly receives entries. This is demonstrated in Figure 6.6, which assumes a flow that transfers entries across three different cores. When the first Peer Space sends an entry to the second one, the associated runtime user *User1* is authenticated using the identity provider (IdP) X. Subsequently, the flow continues in the second Peer Space and causes the transmission of another entry to the third core, which authenticates the runtime user *User2* via a different identity provider Y. Due to the multitenancy support, multiple wiring owners may be involved in the flow on each core. A simple delegation chain is not sufficient, as the level of trust in a subject may also depend on where (and how) its principals have been authenticated and by which other runtimes this information has been forwarded. Therefore, the delegation

chain is extended with separate *authentication chains* for each principal, which include the runtime user identity of the authenticating Peer Space and those of all following cores, where each one has authenticated its respective predecessor and inherited its subject claims. In the example, when examining the principals of a delegation chain at *Peer Space 3*, an involved wiring owner from *Peer Space 1* would be associated with an authentication chain containing *User1*, *User2*, and *User3*.

By providing such detailed subject information, fine-grained trust relationships can be established by means of authorization rules, which are described in Section 6.2.2. Peer owners can decide which forms of delegations are trusted by specifying constraints on delegation and authentication chains. For instance, peers at *Peer Space 3* may accept any delegation received from *Peer Space 2*, while an intermediate peer on *Peer Space 2* may only allow direct access from *Peer Space 1* by a specific wiring owner (i.e., no further indirections). When each participating peer in a flow applies such trust-based rules, an explicit chain of trust is established.

Subject Trees. As delegation and authentication chains for any entry are forwarded along the same path determined by the current flow, they can be combined in a single tree data structure. All authentication chains naturally end with the runtime user of the processing Peer Space, thus it is set as the root of this *subject tree*. An example subject tree is shown in Figure 6.7. Leaf nodes depict principals that participate in the delegation chain, while internal nodes represent runtime users within the authentication chains, whereas each node has directly authenticated all of its children and has been authenticated by its parent. The order of principals within the delegation chain is determined by a left-to-right traversal of the subject tree. Their corresponding authentication chains are retrieved by following the path from the leaf node to the root. While all other principals are represented by their corresponding security attributes, the root is depicted by an empty node, as the identity of the processing Peer Space's runtime user is not relevant for access control within the limits of the local core. In the example, user *Stefan* delegates to user *Eva* at a Peer Space associated with runtime user *SBCServer*. From there, an entry is sent to another runtime (owned by *Server24*), where it is processed by a system user called *Worker1* and finally forwarded to the current Peer Space.

Extending the abbreviated notation for delegation chains, a textual representation of a subject tree can be derived using a simple recursive algorithm:

1. Extract the identities of all direct children of the root node. If more than one exists, they are merged in reverse order using "**for**".

2. Recursively determine the textual representations of the sub-trees rooted at each child (back to step 1) and prepend them to the respective identities of the intermediate nodes using "**@**". If several children exist, parentheses must be used to group sibling nodes together, as the **@** operator has a higher precedence than the **for** operator.

3. For any leaf, the textual representation simply corresponds to its identity.

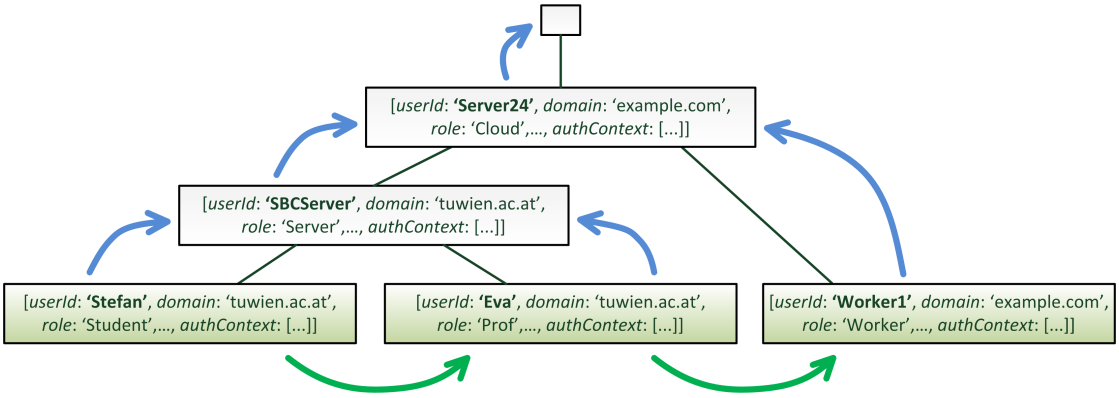

Figure 6.7: Example subject tree with arrows indicating the delegation chain (green) and associated authentication chains (blue)

In the example, the root only has a single child (*Server24*). When evaluating its sub-tree, the textual representation resolves to "(*Worker1* **for** *X*) **@** *Server24*", where *X* stands for the sub-tree with root *SBCServer*, whose evaluation yields "(*Eva* **for** *Stefan*) **@** *SBCServer*". Thus, the complete subject tree is expressed as "(*Worker1* **for** (*Eva* **for** *Stefan*) **@** *SBCServer*) **@** *Server24*". When the complete subject information is required, the user IDs in this notation are replaced by property sets with the corresponding security attributes.

Within an entry, the subject tree can be represented via a nested property that describes all involved principals (see Section A.3.5). This subject property consists of a list of property sets, each of which represents a direct child node of the subject tree root. Besides their respective security attributes, each nested property set may itself have a `children` property that contains a list of property sets for their own child nodes.

Combination of Subjects. Within a flow, the subject trees of the involved entries are iteratively extended. This happens after entries are received from remote Peer Spaces and whenever a wiring emits an entry using indirect access. In the first case, the identity of the authenticated remote runtime user is set as the principal of the previous root node. Then, a new, empty root node is added as its parent, depicting the local runtime user. Using the textual notation, this extension is expressed as *claimedSubject* **@** *authenticatedPrincipal*. In the second case, the subject trees of the wiring owner and the selected input entry owner are merged. As they already share the same root, they can be combined by adding the branches of the wiring owner's subject tree to the subject tree of the delegating input entry owner. In the textual representation, the subjects can be simply joined using a single **for** operator, i.e., *wiringOwnerSubject* **for** *entryOwnerSubject*.

In order to keep subject trees manageable in case of complex interactions, a normalization of the subject tree may be necessary after any combination operation. The general idea is to reduce the number of redundant nodes in the combined subject tree. While some information about the workflow may be lost through this process (e.g.,

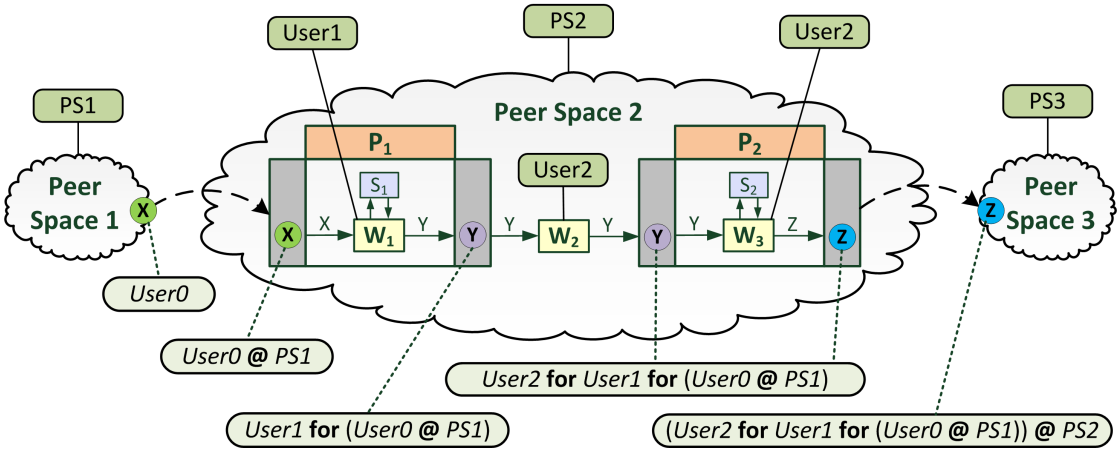

Figure 6.8: Delegation example with subjects at different stages of flow

concerning interaction cycles and execution locations of wirings), the relevant relations among the involved principals are preserved, which allows for trust-based rules matching standardized subject trees. The applicable normalization rules are:

- **Distributivity:** The @ operator can be defined as right-distributive over **for**, i.e., $(X \text{ @ } A)$ **for** $(Y \text{ @ } A) = (X \textbf{ for } Y) \text{ @ } A$, where X and Y can be individual principals or arbitrary sub-trees and A is an authentication chain. Thus, adjacent branches of a subject tree are merged if they have a common authentication chain.
- **Redundancy elimination for delegation chains:** Repetitions of the delegation chain are removed if the consecutive sub-trees are identical (including their authentication chains), i.e., X **for** $X = X$. This often occurs when multiple wirings of the same owner process entries sequentially. From the viewpoint of the authorization process, it is only relevant that the subject was involved at a specific point within the flow, but not how many of its own wirings were invoked.
- **Redundancy elimination for authentication chains:** A similar rule also applies to authentication chains, where consecutive segments are merged if they are identical ($A \text{ @ } A = A$). This usually occurs when entries are emitted to a remote site by wirings owned by the local runtime user.

Figure 6.8 depicts the iterative subject tree extension by means of an example flow involving three Peer Spaces. *User0*, who has been authenticated at *Peer Space 1*, induces the transmission of an entry to a peer at *Peer Space 2*. Upon entering this runtime, the responsible principal for the first Peer Space (*PS1*) is added to the subject tree. The delegation chains of the emitted entries are subsequently extended with the owners of the involved wirings at *Peer Space 2*, which all use indirect access mode. As the same owner is used for W_2 and W_3, the subject does not change for emitted entries of type `Z`. Using the destination mechanism, the result entry is finally sent to *Peer Space 3*, where it is enriched again with the authenticated runtime user of *Peer Space 2* (*PS2*).

6.2.2 Policy Language

Access control for the Peer Model is determined by a set of fine-grained authorization rules, which are specified using a policy language that is based on the one introduced for XVSM. Main differences are the adaptation to the PMQ-based query mechanism and the support for more sophisticated subject templates. Furthermore, the rules do not include a separate field for the effect, as it is implicitly set to `PERMIT`. This greatly reduces the complexity for policy evaluation, as described in Section 6.2.3. In EBNF-like notation, the rule syntax can be defined as follows:

⟨*Rule*⟩ = "`RULE`" ⟨*RuleID*⟩
"`SUBJECTS:`" ⟨*SubjectTmpl*⟩ { "`,`" ⟨*SubjectTmpl*⟩ }
"`RESOURCES:`" ("`*`" | (⟨*Container*⟩ { "`,`" ⟨*Container*⟩ }))
"`OPERATIONS:`" ("`*`" | (⟨*AccessMode*⟩ { "`,`" ⟨*AccessMode*⟩ }))
"`SCOPE:`" ("`*`" | ⟨*Scope*⟩)
"`CONDITION:`" ("`-`" | ⟨*Condition*⟩)

⟨*SubjectTmpl*⟩ = ⟨*NodeTmpl*⟩
| ⟨*SubjectTmpl*⟩ "`for`" ⟨*SubjectTmpl*⟩
| ⟨*SubjectTmpl*⟩ "`@`" ⟨*NodeTmpl*⟩
| "`(`" ⟨*SubjectTmpl*⟩ "`)`"

⟨*NodeTmpl*⟩ = "`*`" | "`**`" | ⟨*PrincipalTmpl*⟩

⟨*PrincipalTmpl*⟩ = "`[`" ⟨*Selector*⟩ { "`,`" ⟨*Selector*⟩ } "`]`"
| "`$$SELF`" *(* matches target peer owner *)*

⟨*AccessMode*⟩ = "`write`" | "`read`" | "`take`"

⟨*Scope*⟩ = ⟨*ScopeQuery*⟩
| "`NOT`" ⟨*Scope*⟩
| ⟨*Scope*⟩ "`AND`" ⟨*Scope*⟩
| ⟨*Scope*⟩ "`OR`" ⟨*Scope*⟩
| "`(`" ⟨*Scope*⟩ "`)`"

⟨*ScopeQuery*⟩ = ⟨*Type*⟩ ["`⟦`" ⟨*Selector*⟩ "`⟧`"]

⟨*Condition*⟩ = ⟨*ConditionQuery*⟩
| "`NOT`" ⟨*Condition*⟩
| ⟨*Condition*⟩ "`AND`" ⟨*Condition*⟩
| ⟨*Condition*⟩ "`OR`" ⟨*Condition*⟩
| "`(`" ⟨*Condition*⟩ "`)`"

⟨*ConditionQuery*⟩ = ⟨*Container*⟩ "`|`" ⟨*PMQ*⟩

The complete syntax specification can be found in the appendix (see Section A.3.7). In the following, the adapted rule elements are described.

Subject Templates. Each rule specifies one or more subject templates, which are matched with the subject tree of the accessing subject. The template itself is represented as a tree of *principal template* nodes, which target the involved principals. Its syntax is based on the verbose version of the textual notation for subject trees. In order to match, a subject template needs to have the same structure as the given subject tree and each of its principal templates must match the principal at the corresponding position in the subject tree.

Principal templates can be compared to subject templates in XVSM, but they provide a higher expressiveness than a simple Linda-based mechanism. They contain predicates on security attributes that are evaluated on the property set of the corresponding principal within the nested subject property. As this property set has the same general structure as a complete entry, PMQ selectors can be reused for that purpose. Selectors specify conditions on specific security attributes, like "*role* = 'Prof'" or "*age* $\geq$ 18". Although such predicates can be combined using logical operators, the rule syntax also allows comma-separated selectors, which represent a conjunction of predicates. A principal template only matches if the examined principal fulfills all of its selectors. This enables a notation that is very similar to XVSM subject templates in simple cases.

For additional flexibility, wildcards may replace principal templates at any position within the template tree. The wildcard for individual nodes ("*") matches any principal at the given position in the subject tree, whereas the second wildcard ("**") matches an arbitrary chain consisting of zero or more principals. Depending on the position in the template, such a wildcard represents either a delegation chain or an authentication chain. In the first case, the wildcard node matches zero or more leaf nodes at the indicated position within the delegation chain. In the second case, the authentication chains are examined separately for each leaf node that is a descendant of the wildcard node, including those represented by delegation chain wildcards. The wildcard stands for zero or more principals within the respective authentication chains. Due to the separate evaluation, the wildcard may actually match different authentication chains for each element of the delegation chain. Therefore, the subject template "** @ **" acts as a general wildcard that matches any subject tree. A more detailed specification of the matching algorithm is given in Section B.2 of the appendix.

With regard to the example subject tree from Figure 6.7, matching subject templates include:

- ([*role* = 'Worker', *domain* = 'example.com'] **for** ([*role* = 'Prof' $\vee$ *age* $\geq$ 30] **for** [*role* = 'Student']) **@** [*role* = 'Server', *domain* = 'tuwien.ac.at']) **@** [*userId* = 'Server24', *domain* = 'example.com']

- ([*role* = 'Worker', *domain* = 'example.com'] **for** (* **for** [*role* = 'Student', *domain* = 'tuwien.ac.at']) **@** *) **@** [*role* = 'Cloud', *domain* = 'example.com']

- (** **for** [*role* = 'Student', *domain* = 'tuwien.ac.at']) **@** ** **@** [*role* = 'Cloud', *domain* = 'example.com']

The first example provides quite specific conditions, which allows for a strict control over the involved principals within a flow. In the second template, the forwarding wiring owner and the first runtime user are not relevant for the applicability of the rule. The last example provides the most general template, as only the originator and the actually authenticated runtime user are specified. The chain wildcards indicate that the student user may have directly sent the entry via the cloud runtime or that several other wiring owners and Peer Spaces were involved in between.

Containers and Operations. As in XVSM, the `resources` field indicates the affected containers, which are specified via their path, i.e., the local container address without the Peer Space URI. The `operations` field is equivalent to the `actions` field of XVSM authorization policies, which has been renamed to avoid confusion with wiring actions. Copy guards require read permissions, while move guards need take privileges. Like in XVSM, take permissions also implicitly grant read access. Write permissions affect direct write operations by actions as well as injection of entries via the API and the destination mechanism. Access to the entry collection requires no authorization, as it only exists within the active wiring instance.

Scope and Condition. Like in XVSM, the scope determines for which kind of entries within the container the rule applies, while the condition restricts the applicability of the rule based on the existence and content of context entries within specific containers. Scope queries are specified using a PMQ variant without a count specification, which is implicitly set to `ALL`. Due to the non-deterministic character of PMQ-based entry selection, other count values would not provide clear semantics. Although logical operators are already supported within selectors, they can also be used to combine multiple scope queries in order to match different types. Condition queries consist of the path of the targeted context container and an associated PMQ expression. The query evaluates to true if a read operation on this container is successful and returns at least one matching entry. Complex conditions are once again possible by joining multiple queries using conjunction, disjunction, and negation.

Dynamic Parameters. In order to also take the request context into consideration, dynamic parameters can be incorporated into authorization rules using the "$" notation. They are not only supported for scope and condition queries, like in XVSM, but also within principal templates. The context is represented by a set of *context variables* that includes the nested subject property (`$SUBJECT`) of the responsible subject. Unlike system variables, which are set by the runtime according to predefined semantics, context variables directly reference corresponding properties of the written entry or the querying wiring, respectively. Thus, also other relevant properties, like the flow ID (`$FID`) of a write operation, are accessible. In contrast to explicit local variable assignments on wiring links, the context is implicitly initialized for each access operation. As they have different ranges of validity, context variables can only be accessed within authorization rules, while local variables are specific to link definitions.

Specific principals within the subject, like the originator and the invoker, are aliased with special identifiers to simplify access. For instance, to declare that a specific principal must come from the same domain as the last element of the delegation chain, the selector "*domain* = $*invoker.domain*" can be added to the corresponding principal template. Within a scope query, the selector "*originator* = $*originator*" indicates that the first element of the accessing subject's delegation chain must be equal to the originator of the flow responsible for writing the examined entry, i.e., only own entries are matched[5]. Another predefined alias is *sender*, which represents the last child of the root node, i.e., the authenticated principal that actually triggers the access. Similar shortcuts can be devised based on the index position of a principal within the delegation chain as well as the index position within a specific authentication chain for inner nodes.

Additionally, system variables like the current time can be referenced within rules. The system variable `$$SELF` has a special role, as it represents a principal template that matches the owner of the affected peer.

Example Rules. Like in XVSM, expressive authorization rules can be formulated, as demonstrated by the following examples that target the management of grades for a lecture on SBC topics:

RULE R1
SUBJECTS: [*role* = 'SBCProf', *domain* = 'tuwien.ac.at']
RESOURCES: *SBCLecturePeer/PIC*
OPERATIONS: *
SCOPE: Grade
CONDITION: *SBCLecturePeer/PIC* | LectureInfo ⟦*lectureEnd* < `$$TIME`⟧

RULE R2
SUBJECTS: `$$SELF` **for** [*role* = 'Student', *domain* = 'tuwien.ac.at']
RESOURCES: *SBCLecturePeer/PIC*
OPERATIONS: `read`
SCOPE: Grade ⟦*studentId* = $*originator.userId*⟧
CONDITION: -

Rule `R1` allows a specific group of professors from TU Wien (with role `SBCProf`) to set and change `Grade` entries in the PIC of the management peer for that lecture. However, this is only possible after the lecture has officially ended according to the `LectureInfo` entry from the same container. The other rule (`R2`) regulates how such grades can be read by TU Wien students. It allows indirect access via an intermediary owned by the peer owner, e.g., by means of an internal wiring that creates dynamic wirings for querying grades based on student requests. Assuming that each grade entry

[5]This assumes that security attributes (including those within the authentication context) do not change between accesses. Otherwise, a consistent subset of the properties has to be compared, e.g., "*originator.userId* = $*originator.userId* ∧ *originator.domain* = $*originator.domain*".

is associated with a specific student number that matches the corresponding user ID, the scope restriction ensures that students can only retrieve their own grades.

6.2.3 Policy Evaluation

For the evaluation of authorization rules, the overall strategy of the XVSM access control model is followed. At first, the rule target is examined. The accessing subject has to match at least one specified subject template, the targeted container must be listed, and the used operation must be compatible with the rule. For each matching rule, the respective condition (if specified) is subsequently evaluated. If it is satisfied, the rule applies to all entries that match the scope (or to the whole container if no scope is given). However, the evaluation process can be simplified compared to XVSM due to two important differences in the policy language. The lack of coordinators enables more efficient scope matching, while the omission of rule effects allows for implicit rule combination.

Instead of having to use authorization-aware queries on the whole container, the scope queries can be evaluated for each entry individually as Peer Model containers do not provide a specific order or other dependencies among entries. Read and take operations only select entries that match both the guard's PMQ and the scope of an applicable rule. Thus, transparent access is ensured, as only permitted entries are visible for wirings. For write operations, authorization can be checked in advance by evaluating if all given entries match the scope of an applicable rule.

As for XVSM, access decisions are determined at the level of entries. However, no complex combination algorithm is required if multiple rules apply for a container access operation. Any entry that is covered by at least one applicable rule is permitted, while the remaining entries are implicitly denied. Rules that could not be evaluated (e.g., due to invalid dynamic parameters) can safely be ignored as they do not affect the privileges granted by other rules.

6.3 Secure Peer Space Architecture

For the enforcement and administration of authorization policies, the concepts of the Secure Service Space can be combined with the meta model approach of the Peer Model. In contrast to XVSM, the complete middleware functionality can be mapped to container access on user and system peers. Therefore, there is no need for additional task or answer containers. However, the hierarchical structure of peers with possibly different owners has to be considered. As sub-peers are logically embedded in their respective parent peers, access shall only be possible if corresponding permissions for parent and ancestor peers exist. These constraints are addressed by the generic Secure Peer Space architecture, which is depicted in Figure 6.9.

Remote clients inject entries into containers of the Runtime Peer. In most cases, these entries enter the space via the RTP's PIC, from where they are forwarded to a specific sub-peer via an explicit wiring or based on their destination properties. They are

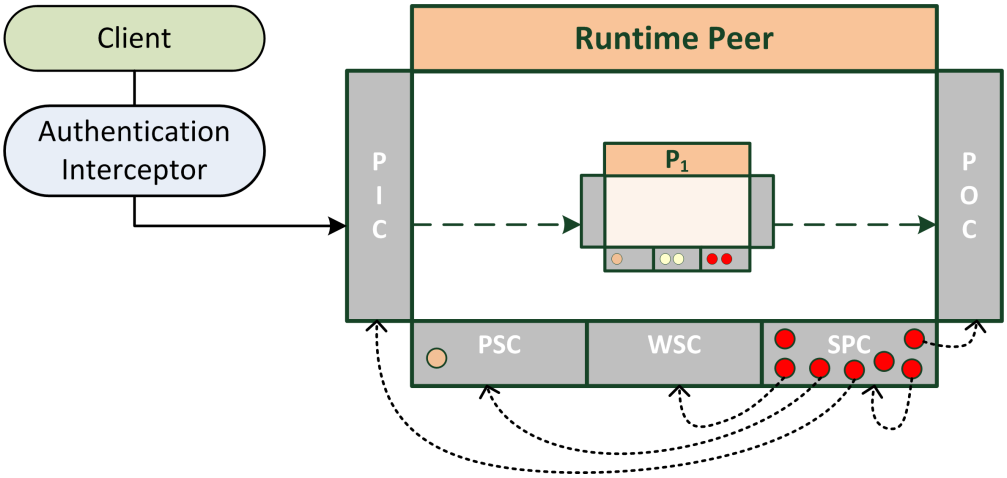

Figure 6.9: Generic Secure Peer Space architecture

authenticated via an exchangeable authentication interceptor, which verifies the provided credentials and sets the entries' subject properties accordingly. Any container access triggered by these entries has to be authorized by the Peer Space, using the respective responsible subject according to the subject concept described in Section 6.2.1. Like in the Secure Space, authorization rules are managed in a bootstrapped way in the form of *rule entries*, which have a special type (`Rule`) and contain the specified rule fields as (nested) properties. However, instead of using a single policy container for the whole space, each peer hosts an additional meta container termed *Security Policy Container (SPC)*, which allows for the management of authorization policies at the granularity of peers.

Rule Specification

Rule entries within the SPC may target any of the peer's containers. PIC rules determine which entries are accepted for handling by the peer's coordination logic, whereas POC rules regulate which kind of entries may be used as output for a peer, thus controlling its potential influence on other peers. Depending on the peer-specific semantics, both containers may cover a wide range of entries that fall into different categories, like requests, stored data, events, responses, exceptions, or context information. In some cases, it may be beneficial to create separate partitions in the form of additional peer containers that group related entries and their security constraints. For instance, by using a distinct state container for storing data entries within a peer, a separation between service-centric and data-centric authorization can be achieved. Administrative rules are possible both for access to the meta model and for the management of the authorization policy itself. Depending on the used subject template, all these rules may cover direct access by external clients (only write), direct access by installed wirings, and/or indirect access by wirings on behalf of other subjects.

The definition of local policies for each peer requires slight adaptations of the rule syntax defined in Section 6.2.2. As the affected peer is implicitly specified by the location of the enclosing SPC, only container names have to be included in the `resources` property instead of full paths. Similarly, also condition queries are only allowed on local containers in order to preserve the encapsulation of peers and prevent covert channels that would allow peer owners to indirectly query entries of co-located security domains without authorization[6]. If context conditions across the peer hierarchy were required, authorization checks for the respective rule owners (i.e., the entry owners of the rule entries) would have to be enforced for each evaluation of a condition, which would cause a significant overhead.

The following rules provide examples for controlling access to meta containers:

RULE DynamicWiringRule
SUBJECTS: [*role* = 'Prof', *domain* = 'tuwien.ac.at']
RESOURCES: *WSC*
OPERATIONS: `write`
SCOPE: Wiring $[\![repeat = 1]\!]$
CONDITION: -

RULE PICAdminRule
SUBJECTS: [*role* = 'Admin', *domain* = 'tuwien.ac.at']
RESOURCES: *SPC*
OPERATIONS: `write`, `take`
SCOPE: Rule $[\![resources = \langle\text{'PIC'}\rangle]\!]$
CONDITION: -

The first rule enables professors to install one-off dynamic wirings, i.e., to write wiring specification entries with a repeat count of one into the peer's WSC. The second rule represents a partial delegation of rights to users with a specific administrator role. This subject may add, update, or delete rules that affect only the peer's PIC, i.e., the `resources` property includes this container as its sole entry.

Hierarchical Policies

The authorization policy of the whole Peer Space is determined by the content of all SPCs within the peer hierarchy consisting of the Runtime Peer and all of its direct or indirect sub-peers. Due to the wiring mechanism, entries within a flow can only be transported up or down along this hierarchy. Therefore, external communication with a sub-peer is only possible via entries that are passed through containers of the parent peer. This holds true for the destination mechanism, whose semantics can be explained via conceptual system wirings that route entries to their intended target containers by recursively traversing the POCs of the sending peer's parents and the PICs of the receiving peer's parents. In

[6]This assumes that policy administrators (apart from the peer owner) at least possess read permissions on all containers of the corresponding peer.

Figure 6.9, they connect the RTP and its sub-peer P_1 in order to depict both inbound and outbound communication. Before any remote request reaches P_1, it is implicitly written to the PIC of the RTP. Similarly, any entry with a remote destination that is emitted by a wiring of the sub-peer has to traverse the POCs of P_1 and the RTP. As such wirings are directly controlled by the middleware runtime, their guards are implicitly authorized, whereas the respective owners of the forwarded entries are used as subjects for the actions. Thus, each peer can indirectly control any external access to or from sub-peers (both via wirings and via the destination mechanism) by specifying corresponding rules in its SPC. The authorization mechanism intercepts flows before entering or after leaving a peer and checks whether the responsible subject is also allowed to access the parent peer's PIC or POC, respectively. In the example, the client user needs write permissions on the PICs of P_1 and the RTP, whereas the responsible subject for emitting an entry from P_1 to a remote destination requires write permissions on both POCs.

This nested policy approach provides multiple layers of protection, as all involved peers within the hierarchy must permit access. In order to realize fine-grained authorization policies with a clear structure and a minimum amount of redundancy, different granularity levels should be used for each level in the peer hierarchy. General, trust-related rules may be feasible for controlling access to the RTP's PIC and POC, whereas permissions may be iteratively refined (e.g., based on entry types, coordination properties, or context conditions) in sub-peers and their children. If a sub-peer does not require any additional access constraints, it can also rely on its parent for access control and permit any non-administrative access via a catch-all rule for its PIC and POC.

Policy Administration

Each SPC is initially managed by the respective peer owner, which has implicit permissions to access all peer containers. As some security attributes (e.g., within the authentication context) may differ between accesses, this applies to any subject with `userId` and `domain` properties that are equal to those included in the peer specification entry. Such a subject can configure wirings with full internal access as well as sub-peers. However, in order to enable a distinction among actions on behalf of different invokers, indirect access (*peerOwner* **for** X) has to be explicitly permitted. Peer owners can add rule entries to grant access to the peer's functionality and possibly delegate some or all of their administrative rights to other trusted subjects.

For the Runtime Peer, the configured runtime user is defined as owner, which enables the bootstrapping of the coordination logic and the authorization policy within the Peer Space. The runtime user adds initial peers and wirings and defines corresponding permissions. If access to the RTP's PSC is granted, other users are able to specify their own peers and manage their authorization policies accordingly via the SPCs of the newly created peers. As they may delegate their administrative rights themselves, added wirings and sub-peers may be owned by different subjects. Thus, nested security domains managed by different administrators are possible within a Peer Space. In order to ensure a strict encapsulation of authorization policies and for privacy reasons, peer owners are not automatically granted permissions on containers of sub-peers. They can, however,

control any incoming or outgoing entries due to the nested policy approach and remove unwanted sub-peers via their PSC.

The runtime user has a special role comparable to a root user in conventional file systems. This subject is implicitly permitted to access any local container. It is also able to impersonate other users within the local Peer Space and operate with their permissions instead. Therefore, it is entitled to use a third access mode besides direct and indirect access, termed *impersonated access*, which enables wirings to set the subject property of an entry to an arbitrary value.

Like for the meta model, the bootstrapped approach via rule entries in the SPC facilitates several useful features for security administration. Rules can be injected from local or remote sources via the destination mechanism. They can be read, updated, or removed using (dynamic) wirings that are linked to the SPC. Policy changes may be triggered directly by an administrator or via predefined wirings that automatically change permissions in reaction to certain events. Furthermore, the lifecycle of authorization rules can be controlled via `TTL` and `TTS` properties.

Exception Handling

Authentication and authorization errors lead to the creation of corresponding exception entries, which are stored in the peer where the access violation occurs. Their target container depends on the configuration of this peer. They may be stored in the POC or a separate exception container. Exception entries are owned by the runtime user, thus their creation is implicitly permitted. They contain relevant properties of the denied operation, including its parameters and the responsible subject. Exception handling can be performed by installing wirings that process such exception entries. As the overall authorization mechanism also applies to these entries, only specific subjects may access them. If there are no wirings for treating exceptions locally or sending them back to the invoker, the entries should expire after a short time according to their `TTL`.

6.3.1 Integration into Peer Space Runtime Architecture

The described access control concepts of the Secure Peer Space are realized by extending the Peer Space runtime architecture with corresponding authentication and authorization features. In contrast to XVSM, both local and remote access have to be authorized. However, internal access by the middleware runtime itself (e.g., for monitoring the meta containers) is always permitted. Authorization is required whenever wiring guards are evaluated, wiring actions are executed, or entries are injected by applications. Thus, invocations of the container middleware have to be intercepted accordingly in order to enforce the active authorization policy. Policy management can be enabled via a natural extension of the meta model by adding an SPC to each peer.

The runtime user is determined by the hosting application of the Peer Space, which is responsible for setting the initial coordination logic and corresponding permissions. In a distributed application, each involved principal actually represents a local user at a specific Peer Space, i.e., either the corresponding runtime user or a locally defined system

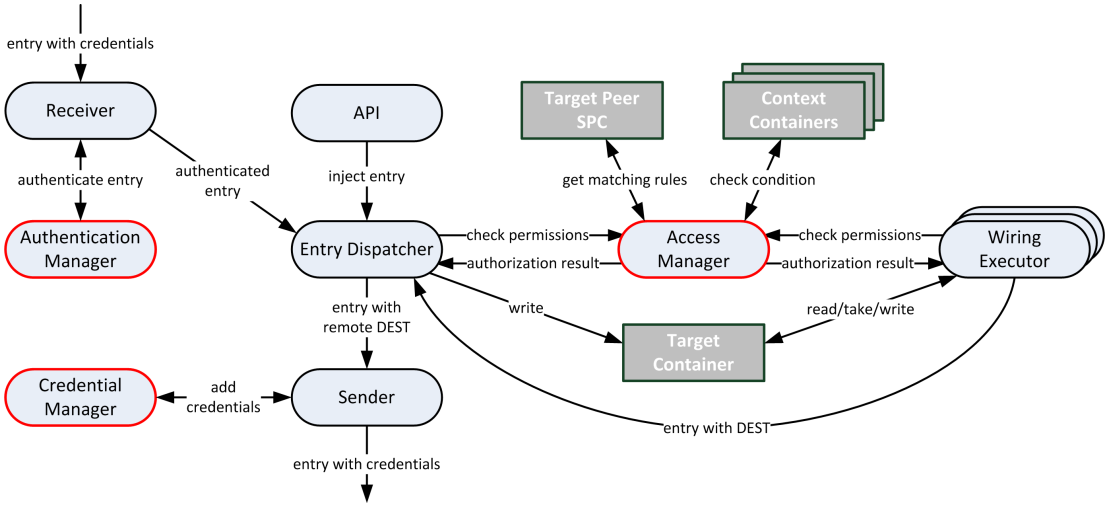

Figure 6.10: Secure Peer Space access control architecture

user with restricted privileges. Following the P2P paradigm, a Peer Space can act as a client, a server, or both, depending on its coordination logic. Therefore, the middleware must not only correctly authenticate other Peer Space instances, but also authenticate itself at remote runtimes.

In the following, the extended runtime architecture for the Secure Peer Space is described.

Access Control Architecture Overview

Figure 6.10 shows how the envisioned access control mechanism can be integrated into the simplified Peer Space runtime architecture described in Section 6.1.3. It resembles the approach used for XVSM, but is not restricted to a specific container middleware.

Received entries are authenticated using an *authentication manager*, which verifies the included credentials and sets the subject property accordingly. To enable authentication at the remote target Peer Space, the *credential manager* enriches outgoing entries with the credentials of the associated runtime user. When the local API is used by the hosting application, the authentication mechanism is bypassed and the runtime user is implicitly set as subject using a special principal with ID “RuntimeUser”. Thereby, authorization rules need not know the actual identity of the local runtime user. In order to enable multiple local system users with limited permissions, the subject property may be defined explicitly instead. As the runtime user can impersonate arbitrary subjects within the local space (either directly or by installing wirings that apply impersonated access), only fully trusted entities should be able to access the API.

Like in XVSM, a dedicated *access manager* component acts as PDP and evaluates access requests according to the active authorization policy. It is invoked both by the entry dispatcher and by the wiring executor before they access any container. They enforce the obtained access decision when executing the respective container access

operations and thus form the PEPs. The meta model listener is not depicted as it only performs internal operations that are implicitly permitted. PAP and PIP are again bootstrapped by means of containers.

In the following, the components of this access control architecture are described in more detail.

Authentication Mechanism

For every authentication process, two Peer Space cores are involved: the sending runtime and the receiving runtime. As environments with different identity providers are possible, the authentication and credential manager components may support multiple authentication mechanisms (e.g., passwords, certificates, SSO), which can be dynamically selected based on the current communication partner.

For outgoing entries, the sender component invokes the credential manager before transmitting to the remote site, which retrieves the preconfigured credentials of the runtime user and adds them to the entry via the `CREDS` property. This includes claimed security attributes of the sending runtime, like its user ID and domain. In order to enable authentication at different runtimes with non-overlapping identity providers, multiple identities with corresponding credentials may be used, which are dynamically selected according to the destination target site.

The receiver component forwards each incoming entry to the authentication manager, which then extracts the credentials from the corresponding property. Based on the specified domain and the used credential format, the authentication manager selects the associated identity provider and subsequently verifies the credentials. Finally, the entry's subject tree (i.e., its `SUBJECT` property) is enriched with the security attributes of the authenticated principal as described in Section 6.2.1, whereas additional information about the authentication process is included in the `authContext` field. Whenever the special *RuntimeUser* principal is encountered within the claimed subject, it is automatically replaced by the identity of the authenticated principal, which enables subsequent simplification of the subject tree via the normalization procedure (e.g., $\mathit{RuntimeUser}\ @\ \mathit{PeerSpace1} = \mathit{PeerSpace1}\ @\ \mathit{PeerSpace1} = \mathit{PeerSpace1}$). If authentication fails, the corresponding entry is discarded. An authentication exception entry is generated instead and stored within the RTP.

Subject Handling

The correct and safe handling of the subject property within a Peer Space after the authentication process requires some additional adaptations. To prevent bypassing of the access control mechanism via manipulation of the subject property within a wiring, the subject is specified as a hidden system property. Such properties can only be accessed by the runtime itself and not by services or links, except when the runtime user is the corresponding wiring owner.

For realizing the delegation mechanism, an additional coordination property is defined that can be set by services and link assignments for emitted entries. This delegation

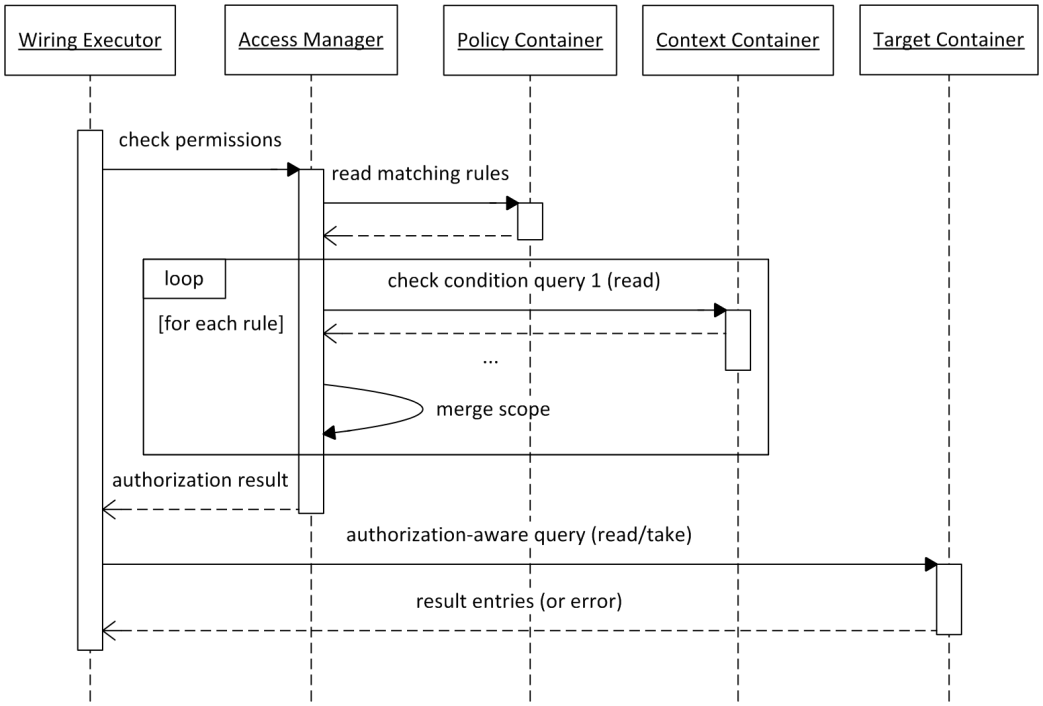

Figure 6.11: Secure Peer Space authorization workflow for guards

property (`DLG`) references the entry ID of one of the input entries, whose associated subject shall be used as delegator. When the associated action is processed, the runtime retrieves the subject from the selected input entry and merges it with the wiring owner in order to form the owner of the emitted entry. If the `DLG` property is not defined, direct access mode is used instead and the wiring owner is set as the subject. Impersonated access by the runtime user is indicated by the `IMP` flag. If it is set, the runtime does not change the value of the emitted entry's subject property.

Authorization Workflow

The authorization workflow is shown for guard access in Figure 6.11 and for action access in Figure 6.12. In contrast to XVSM, all operations can be checked in advance and the access manager does not have to query the target container directly.

Before the access manager is invoked, the wiring executor initializes the request context. It contains context variables based on the entry to be written or the querying wiring (including the responsible subject) as well as system variables set by the runtime, which enables the resolution of dynamic parameters during the evaluation of the authorization rules. As entries emitted by an action may have different owners, authorization has to be checked separately for each entry, although certain optimizations are possible to prevent redundant computations (e.g., by caching condition results).

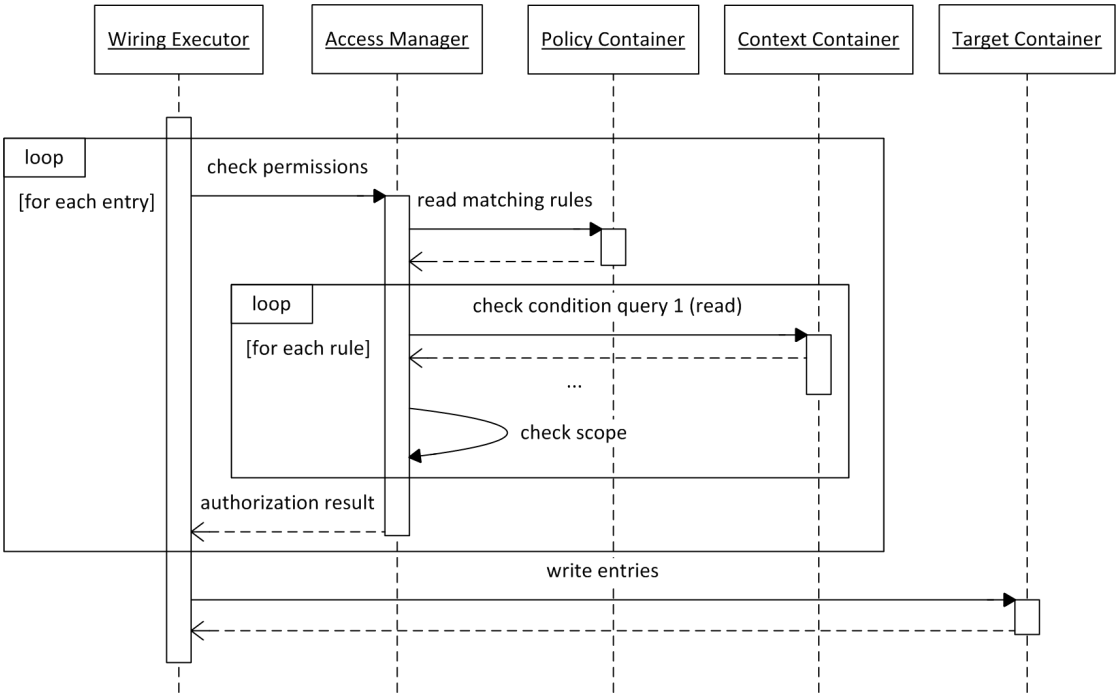

Figure 6.12: Secure Peer Space authorization workflow for actions

When called, the access manager first retrieves relevant rules with matching target. For this, it selects all rule entries with matching `operations` and `resources` properties from the SPC of the peer where the accessed container resides (e.g., using the query "Rule [ALL] $[\![(resources = \texttt{NULL} \vee \text{'PIC'} \in resources) \wedge (operations = \texttt{NULL} \vee \text{'take'} \in operations)]\!]$"). As it cannot be expressed by means of a simple PMQ, the subject template matching algorithm (see Section B.2) is subsequently executed by the access manager to filter out the remaining rules that are relevant for the current subject. The applicability of each rule is then checked by evaluating the condition via read operations on the specified context containers. Finally, the scope has to be considered for rules with satisfied conditions.

For read and take access via guards, the scope is not directly evaluated within the access manager. Instead, the scopes of all applicable rules are merged into a single *authorization scope* using disjunction, e.g., "*scopeRule1* `OR` *scopeRule2* `OR` *scopeRule3*". Therefore, the combined predicate evaluates to true for all permitted entries within the target container. The access manager returns the computed access decision to the wiring executor. If no applicable rule could be found, the decision is `DENY` and the operation has to be canceled. If at least one rule with a wildcard scope applies, access to all container entries is allowed, which is indicated with the return value `PERMIT`. Otherwise, the decision is set to `RESTRICTED-PERMIT` and the computed authorization scope is included in the returned result. For enforcing the authorization result, the wiring executor combines the obtained authorization scope with the PMQ of the examined guard.

Therefore, the PMQ selector is merged with the authorization scope via conjunction, i.e., "*selector* ∧ *authorizationScope*". To enable this operation, the authorization scope has to be transformed first by integrating the type information directly into the selectors of the involved scope queries. For instance, a scope of "A `OR` B $[\![val > 10]\!]$" is changed to "$(TYPE = \text{'A'}) \vee (TYPE = \text{'B'} \wedge val > 10)$", which is valid syntax for a PMQ selector[7]. The wiring executor then performs the corresponding read or take operation on the target container using this extended PMQ, which ensures authorization-aware queries that may only access permitted entries. In combination with the non-deterministic query semantics, this mechanism provides transparent access for wirings, which operate on restricted views of their input containers.

For write operations, the access manager evaluates the scope predicates on the entries to be written. If the specified entry is covered by the scope of at least one applicable rule, the returned decision is `PERMIT`. If no matching rule is found, a result of `DENY` is returned. The action can only be performed if each write operation is permitted, otherwise it is canceled by the wiring executor.

A similar workflow is also used for entries written by the entry dispatcher. However, the access manager has to be called not only for the final target container, but also for the intermediate containers according to the routing semantics of the destination mechanism. Instead of using explicit system wirings and enforcing authorization for each of their actions, the entry dispatcher simulates them by invoking the access manager for each POC and PIC on the entry's path within the local space. The appendix (Section B.3) provides a corresponding algorithm specification. If any of these authorizations fails, the dispatching process is stopped. This allows for a more efficient solution because authorization can be checked in a single step and entries do not actually have to be written into intermediate containers.

Whenever a `DENY` decision is obtained, the corresponding PEP has to handle the authorization error. For write operations, the link execution is stopped and the respective entries are discarded. Depending on certain configuration parameters, which are not detailed here, failed links may be skipped or cause a rollback of the entire wiring instance. However, this only applies to write operations on action target containers by the wiring executor. Subsequent authorization checks by local or remote entry dispatchers according to a set `DEST` property are performed asynchronously and do not affect the successful termination of the wiring instance. For read and take access, evaluation of the corresponding wiring is suspended as it cannot be triggered at the moment. In order to enable developers to react to such errors, an exception entry is generated by the PEP and stored within the peer where the authorization error occurred.

Policy Administration

Rules can be set by injecting entries with type `Rule` into the SPC of the targeted peer. Their internal structure is described in the appendix (Section A.3.8). The runtime must

[7] As an optimization, the targeted entry type for the evaluated query operation could be considered in order to simplify the authorization scope. For instance, if the guard queries entries of type `B`, the authorization scope in the example could be shortened to "$val > 10$".

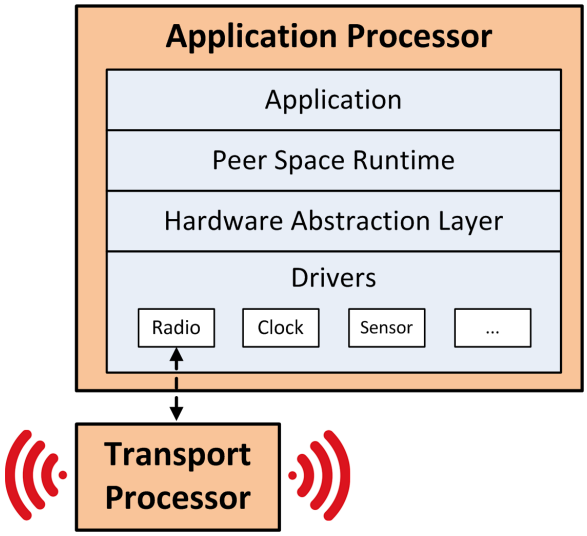

Figure 6.13: LOPONODE system architecture with Peer Model support

ensure that only valid rules with a unique ID are written to that container. This key facilitates simple policy management, as it can be used within a PMQ selector of a wiring guard to update or remove the rule.

6.4 Security Model for Wireless Sensor Networks

Embedded devices for wireless communication — as used within the LOPONODE project (see Section 3.1.3) — have different requirements than general-purpose computers. Instead of relying on an established network topology like the Internet, nodes in a WSN have to form their own network that ensures the correct routing of messages. While the Peer Model is suitable for modeling coordination logic within a WSN, severe resource and timing constraints restrict the set of practically feasible features that can be supported by Peer Space runtimes on the involved embedded devices. Therefore, a scaled-down Peer Space variant is required, which also includes significant changes to the previously defined access control model. The following paragraphs first outline the characteristic features of this restricted Peer Space version and then suggest a suitable security concept. Details of the envisioned access control mechanisms are not fully specified, however, as the fine-grained access control features were not relevant for the examined reference use case and have therefore not been implemented.

Figure 6.13 shows the system architecture for the integration of the Peer Model with the LOPONODE hardware. The application-specific communication protocol is modeled with a restricted version of the Peer Model, which runs on top of an embedded middleware runtime representing the Peer Space. For platform-independent communication with the application processor, a hardware abstraction layer is used that interacts with the hardware-specific system drivers. This includes communication with a transport processor

that actually transmits and receives messages using radio communication. The *Embedded Peer Model*, which is described in more detail in [Ham15], has the following distinguishing properties:

- Each node runs only a single peer, i.e., there are no sub-peers. Containers beside PIC and POC are not supported.
- Dynamic changes via the installation of wirings are not possible. The entire node behavior has to be specified at design time using a DSL that is automatically translated into executable code by a special compiler tool.
- Exception handling is not supported.
- Nodes are addressed via their unique node ID. Alternatively, a special `DEST` value can be set to enable broadcasting to all nodes in range.

Using the Embedded Peer Model, event forwarding protocols for the railway crossing use case can be specified [KCJN14, KCH14]. In a similar way, also applications that involve multiple stakeholders can be designed. If fine-grained access control is required, the previously described Secure Peer Space architecture has to be adapted:

- Due to the lack of dynamically injected sub-peers and wirings, there is no need for a meta model and all wirings of a node are implicitly owned by the configured runtime user.
- To keep the access control model simple, only external write operation on the PIC have to be authorized, which corresponds to received radio transmissions from remote nodes. Internal access on PIC and POC by wirings is always permitted. Therefore, the `operations` and `resources` fields can be removed from rules.
- To enforce memory and performance restrictions, the complexity of `scope` and `condition` fields as well as subject templates may be limited (e.g., only allowing a type-based scope, a single condition query without dynamic parameters, or principal matching via equality checks). Authenticated security attributes are restricted to an ID as well as optional role and organization fields.
- Wirings can still decide between emitting entries with direct or indirect access mode. However, as there is only a single user per node, subject trees can be replaced by simple delegation chains. The required memory for these delegation chains must be bounded, though. Typical use cases only require constraints based on the actual invoker and the originator of a flow. Therefore, the identities of intermediate nodes can be dropped. Thus, a subject template of "[*id* = 'LN1', *org* = 'X'] **for** [*id* = 'LN2', *org* = 'Y']" in the embedded version approximately translates to "(** **for** [*id* = 'LN2', *org* = 'Y']) **@** ** **@** [*id* = 'LN1', *org* = 'X']" in the full Secure Peer Model notation.

- Like peers and wirings, authorization rules are statically defined via the DSL. They do not have to be managed in a separate meta container.

Authentication is realized by means of shared keys and symmetric encryption with AES [Nat01], which is directly supported by the LOPONODE hardware. A message authentication code (MAC) is attached to each entry, which can be viewed as a cryptographic hash value of the whole message (including meta data) based on a secret symmetric key. The receiver can then recompute the MAC of the received message and compare it with the attached MAC using the same key. If the MACs are equal, the receiver can be sure that the sender is valid (i.e., it has knowledge of the specific secret key) and that the message has not been manipulated. Thus, the locally stored subject information linked to the key can be injected into the entry. As the authentication mechanism is fixed, no authentication context needs to be set in the entry's subject property. Additionally, the MAC acts as a checksum that enables detection of transmission errors. Optionally, each entry may also be encrypted for confidentiality reasons (preferably using a different key).

In order to uniquely identify the sender of an entry, pairwise keys need to be established that are shared with only one other node, respectively. However, this approach does not permit efficient broadcast transmissions, as a message would have to be resent with different MACs for each potential receiver. Therefore, also group keys can be supported that are shared among multiple nodes. In this case, the predefined group ID is set as the entry owner, as the actual sender within the group cannot be verified.

Unlike in the original Peer Model, direct communication between arbitrary nodes via secure channels cannot be assumed. There may be scenarios where two LOPONODEs want to communicate securely via a network of other LOPONODEs that they do not fully trust. The original security model does not support this, as it requires a chain of trust between the originator and the final destination. If such a trust relationship is not feasible, an additional end-to-end security mechanism can be used for secure routing between two LOPONODEs. This can be achieved via nested MACs that use different keys. The originator A and the intended final receiver B may share a pairwise key K_1, while the nodes responsible for routing (as well as A and B) share a group key K_2 (with associated identity G). A verified form of delegation can now be achieved by applying a MAC using K_1 to the original (optionally also encrypted) entry, and then generating another MAC of the whole message (including the first MAC and additional properties necessary for the routing protocol) with K_2. The originator emits an entry with subject "G **for** A" and both MACs included in the credentials field. For the routing nodes, the outer MAC can be verified to authenticate the sender as a member of group G. As each routing node shares the same group identity, the subject remains the same until the message reaches the endpoint. There, both MACs can be validated and the subject "G **for** A" can be confirmed. To indicate that a delegator has been authenticated, the runtime sets a special flag as an additional principal property. This enables subject templates to distinguish between regular and validated delegation.

For an event forwarding scenario, where a sensor node transmits messages to an actuator node via a chain of forwarder nodes, a simple authorization strategy may use the following rules:

RULE RoutingRule
SUBJECTS: [*id* = 'WSN1'] **for** **
SCOPE: *
CONDITION: -

RULE EventRule
SUBJECTS: [*id* = 'WSN1'] **for** [*role* = 'Sensor', *org* = 'X', *verified* = `true`]
SCOPE: SensorEvent
CONDITION: -

The first rule is installed at forwarder nodes and allows free communication within the network defined by the group key, while the second rule applies to the actuator node and allows only sensor event entries that originated from a valid sensor node. Due to the previously described secure routing mechanism, compromised forwarder nodes cannot invent sensor events. However, additional measures are necessary to prevent replay attacks, where old messages with valid MAC are simply repeated. This may include sequence number checks, time synchronization, and key renewal mechanisms, which could be bootstrapped via special wirings. As wireless communication is inherently unreliable, the safety of a WSN application has to be ensured at the application level via corresponding mechanisms that reduce the number of lost messages and reliably detect communication faults. For the railway domain, suitable safety measures are suggested in [OA11].

6.5 Implementation

The Peer Model specification and its Peer Space middleware are still in the development stage. Therefore, no fully featured implementation has been published yet. However, it is planned to create several open source versions for different platforms, which are compatible with each other via an interoperable protocol. The current research prototype [Cej19] is written in Java, while an earlier version with restricted functionality was realized for the .NET framework using C♯ [Rau14]. Other implementations were created for mobile devices based on Android [Sch17, Til17] and for resource-constrained embedded devices [Ham15] (see Section 6.4). Additional prototypes for further programming languages (e.g., Go) as well as necessary development tools like a graphical modeler and a simulation tool are currently planned or in development.

In order to validate the described access control model and the corresponding Secure Peer Space architecture, the introduced security concepts have been integrated into the Java and .NET prototypes of the Peer Space. In the future, also other versions may incorporate these features. This includes the current embedded version, which could be extended with the specialized security model from Section 6.4.

6.5.1 Java Prototype

The Java implementation of the Peer Space is mostly based on the original Peer Model concepts [KCJ+13, KCJN14, KCH14]. Link assignments are not implemented, but can be simulated via services. In addition, also the meta model extensions described in Section 6.1.2 are supported, which are a prerequisite for the realization of the Secure Peer Space. However, wirings do not have repeat count properties. Instead, one-off wirings require a guard that removes their own wiring specification entry from the WSC. Compared to the Peer Model description from Section 6.1, some additional deviations exist that also affect the access control mechanism. The coordination logic is realized via a hierarchy of `Peer` instances, which store references to their sub-peers, wirings, and containers. These peers also integrate the wiring executor logic with separate threads for testing guards and executing triggered wirings. Local entry dispatching is realized by recursively calling the `addEntry` method on peers along the path to the specified destination. Lambda functions, which were introduced in Java 8[8], are used as query language instead of PMQs. This provides an even higher expressiveness, as arbitrary filters can be defined that depend on coordination properties of entries within the examined container.

In [CKSW17], an extended version of MozartSpaces with advanced lifecycle management and eventing features was presented as a suitable candidate for a container middleware within the Peer Space runtime. However, in order to reduce the complexity of the solution and to improve its performance, a lightweight container implementation based on standard maps was used instead, whereas transactional access is ensured via a simple locking scheme based on container-level locks. Due to this design decision, the security extensions implemented for MozartSpaces (see Section 5.5) could not be reused and the suggested access control mechanisms had to be implemented from scratch on top of the current Java Peer Space prototype (see [Cej19] for more details on the non-secure middleware runtime). With few exceptions (e.g., the missing support for the dynamic selection of identity providers), all specified features have been realized. In the following, relevant parts of the Java Secure Peer Space implementation are described[9].

Policy Specification. Authorization rules are `Entry` objects with specific type and coordination properties. They are written to a peer's SPC, which is realized as a special meta container whose content is automatically synchronized with a rule list in the corresponding `Peer` instance. In order to ease the specification of such rules, they can be instantiated via special builder objects that provide a so-called fluent interface. Instead of using complex constructors, objects are iteratively defined by chaining together several simple configuration methods that each return a modified builder instance.

Listing 6.1 shows the creation of an example rule via the `RuleEntryBuilder` class. Its constructor only requires a rule ID and it initially represents an empty default rule. Before the corresponding rule entry is created using the `build` method, the builder is

[8] `https://docs.oracle.com/javase/8/`, accessed: 2020-04-09

[9] An open source version will be available via `http://www.complang.tuwien.ac.at/eva/`.

refined by specifying two subject templates, the target container, the affected operation, scope and condition queries, as well as a destination parameter (line 8). The `addEntry` method (line 9) finally injects the created rule entry into the desired target container (*P1/Sub1/SPC*). Selectors for the scope and condition queries are specified using the functional interface features of Java 8. An `EntryPredicate` defines a Boolean function with two input parameters for the currently evaluated entry (`e`) and the associated request context (`c`). Thus, arbitrary scope and condition queries are possible that also support dynamic parameters via the inclusion of context properties. The example realizes the scope query "X $[\![user = \$originator.userId]\!]$" (lines 1–2), whereas the condition query (line 4) does not specify a predicate and simply checks for an entry of type `Y`.

Listing 6.1: Rule specification in Java Secure Peer Space prototype

```
1 EntryPredicate scopeSel = (e, c) -> e.getUserCoData("user").
    equals(c.getSubject().getOriginator().getUserId());
2 Scope scope = new Scope(EntryType.getEntryType("X"), scopeSel);
3
4 ConditionQuery cq = new ConditionQuery(Address.PICAddress,
    EntryType.getEntryType("Y"), null);
5
6 ... // define subject templates
7
8 Entry rule = new RuleEntryBuilder("Rule1").
    subject(subjTmpl1).subject(subjTmpl2).
    resource(Address.PICAddress).operation(Permission.WRITE).
    scope(scope).condition(cq).
    dest(new Address(
      new PeerAddress("P1/Sub1"), Address.SPCAddress)).
    build();
9 model.addEntry(rule);
```

Subject templates are complex objects themselves and are thus also instantiated via a builder object. Listing 6.2 shows the definition of an example subject template via the `SubjectTemplateBuilder` class. At first, the involved principal templates are specified. When simple template matching is sufficient, a `Principal` can be used, where only relevant fields are set. The first template references a specific user via its ID and domain (line 1), whereas the second template matches all users from the given domain with an admin role (lines 3–5). More complex constraints can be represented using the functional interface `PrincipalPredicate`, which enables arbitrary predicates based on the evaluated principal (`p`) and the request context (`c`). The third subject template in the example provides such a constraint, where only adult users with client role are matched (line 7). Principal templates and corresponding wildcards are then combined in a `SubjectTemplate` object via the builder (line 9). For the delegation chain, static methods (`user`, `any`, `anyMult`) are used to initialize the respective nodes, which can then be merged via `forSubject`. The corresponding authentication chain

elements are added via the functions `authAtUser`, `authAtAny`, and `authAtAnyMult`, which are applied to the respective preliminary builder object. Thus, the example corresponds to the subject template "([*role* = 'Admin', *domain* = 'example.com'] **for** [*role* = 'Client' ∧ *age* ≥ 18]) @ ** @ [*userId* = 'Server1', *domain* = 'example.com']".

Listing 6.2: Definition of subject template in Java Secure Peer Space prototype

```
1  Principal serverTmpl = new Principal("Server1", "example.com");
2
3  AttributeMap securityAttributes = new AttributeMap();
4  securityAttributes.put("role", "Admin");
5  Principal adminTmpl = new Principal(null, "example.com",
       securityAttributes, null);
6
7  PrincipalPredicate clientTmpl = (p,c) ->
       p.getSecurityAttribute("role") != null &&
       p.getSecurityAttribute("role").equals("Client") &&
       p.getSecurityAttribute("age") != null &&
       (Integer) p.getSecurityAttribute("age") >= 18;
8
9  SubjectTemplate subjTmpl1 =
       SubjectTemplateBuilder.user(adminTmpl).
       forSubject(SubjectTemplateBuilder.user(clientTmpl)).
       authAtAnyMult().
       authAtUser(serverTmpl).
       build();
```

Policy Enforcement. Following the general architecture of the Java prototype, the access manager logic is directly integrated into the `Peer` class by adding two additional methods that are called for authorization checks within the entry dispatching and wiring execution logic. The `checkWriteAccess` function tests if a given entry can be written to a specified container according to the current policy. It returns `true` if the operation is permitted and `false` otherwise. The `checkGuardAccess` method decides authorization for a specific container depending on the used operation type (`read` or `take`). It returns an `EntryPredicate` that represents the computed authorization scope. When access is denied completely, a `null` value is returned instead. In case of a general permit, a simple predicate that is always satisfied is returned, i.e., "`(e, c) -> true`".

For both methods, the relevant context information, which includes the responsible subject, is passed via an additional parameter. Instead of querying the policy container for each invocation, rules are directly read from the stored rule list in the `Peer` instance. With this approach, applicable rules can be easily filtered by the access manager methods using the functionality of Java 8 streams, whereas condition queries are tested by evaluating temporary guard links on the respective containers.

Authentication and Subject Handling. In order to enable support for different identity providers, the authentication and credential manager components are implemented in a generic way. The concrete mechanism is masked by two interfaces. The `CredentialVerifier` interface includes a method that returns a `Principal` object (or raises an exception) based on the provided credentials, while the `CredentialProvider` interface determines a method for retrieving the credentials of the local runtime user. For the Java prototype, a simple password-based authentication has been implemented, where credentials and security attributes are stored in local files. An alternative authentication mechanism via certificates issued by a trusted CA is described in [Let18], which also introduces encrypted communication via TLS.

For entries that are emitted by an action, the `setEntryOwners` function sets the respective subject property. It is called by the wiring execution logic right before the entries are written to their target container and the corresponding permissions are checked. When indirect access mode is selected, this method also ensures that the subject of an actual input entry is selected as delegator.

Integration of the Security Architecture. When a Peer Space instance is configured to run in secure mode, authentication and authorization features are automatically enabled during startup of the runtime. Important parameters like the preferred authentication mechanism or the location of a local user database can be set in a corresponding configuration file.

6.5.2 .NET Prototype

The .NET implementation of the Peer Space provides a limited set of coordination features compared to the full model. There are no nested sub-peers, wirings cannot be installed remotely, and multitenancy is not supported. Therefore, also the security concept has been simplified in the corresponding prototype extensions described in [Bit15].

As the respective runtime user is responsible for all peers and wirings of each space, a differentiation between delegation and authentication chains is not necessary (like in the embedded version). Therefore, subjects and their templates are expressed solely via delegation chains. For the same reason, local access by the runtime user does not need to be authorized and only write operations from remote spaces are checked by access control. Similar to the Java prototype, `scope` and `condition` fields are realized via lambda functions, while for subjects only template matching is supported.

Authentication in this version is based on a PKI with a central identity provider that verifies all requests. As no meta containers are supported by this implementation, rule management is performed via a special system peer that is responsible for the entire space. Although not all concepts of the described access control model are realized in this version of the Secure Peer Space, it provides an additional proof of concept for the feasibility of the approach and its adaptability to different profiles of the Peer Model.

6.6 Benchmarks

Analogous to the analysis of the XVSM implementation MozartSpaces, simple micro benchmarks have also been performed for the Java implementation of the Secure Peer Space in order to determine the impact of the access control mechanisms on performance and scalability. They target basic middleware features, namely the execution of wirings and the forwarding of entries between runtimes.

6.6.1 Test Setup

The benchmark framework from Section 5.6 has been adapted for the Java Secure Peer Space. The overall test approach with several warmup and benchmark rounds on a single machine is retained (with a maximal standard deviation of 6%), while the hardware and JVM configuration is the same as for the MozartSpaces benchmarks. Besides adapting relevant security parameters for each test run, the default configuration of the middleware runtime is used.

As before, the focus lies on authorization, while authentication is only simulated. The first test scenario addresses local authorization within a single core, whereas the identity of the respective wiring owner is injected via the API. For the second benchmark series, entries from foreign cores are authenticated via a dummy authentication module, which accepts any provided identity without verifying the credentials. Runtime users are identified by their user ID and domain, while the actual actors additionally have role and age attributes.

6.6.2 Wiring Execution Benchmarks

In this scenario, a wiring repeatedly takes and writes entries within a peer. The performance of local wiring execution and its scalability with the number of entries in the accessed container is examined. Before the start of the benchmark, the peer's PIC is filled with a configurable number of entries and the wiring is injected into the peer. For each iteration, the wiring takes an arbitrary entry with a specific type that is matched by half of all entries. The retrieved entry is then written back to the PIC using indirect access mode, so that the total number of entries within the container remains constant. The middleware runtime has to ensure that the take and write operations by the wiring are authorized. The benchmark round is stopped when 10,000 invocations of the wiring have been reached.

Figure 6.14 shows the benchmark results with different authorization policies of increasing complexity. The baseline case with deactivated security features exhibits only little influence by the number of entries, as only a single matching entry has to be found in any case. In the "owner authorization" configuration, the wiring has the same owner as its enclosing peer and the entries. Thus, all access is implicitly permitted and no rules have to be defined. This shows a basic access control overhead of 10–12% for any entry count.

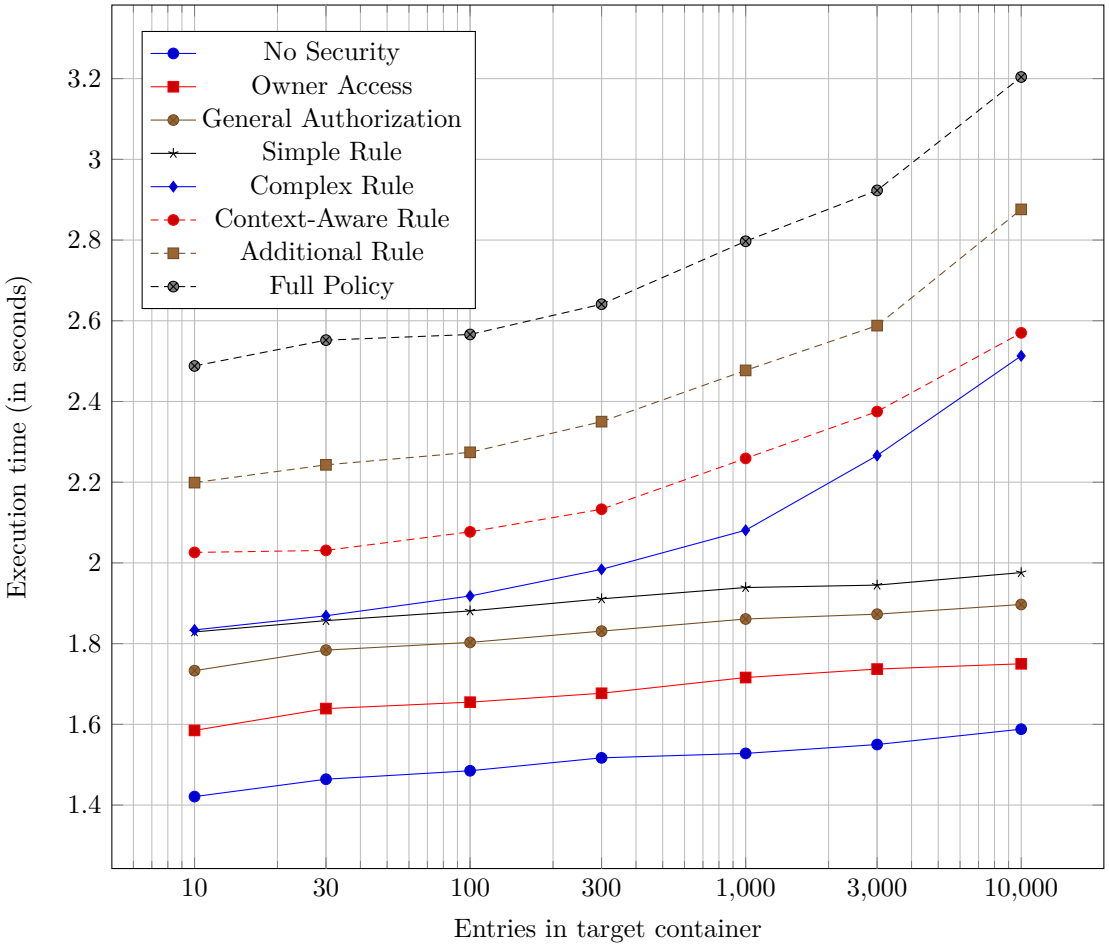

Figure 6.14: Wiring execution benchmarks with 10,000 iterations

For the remaining configurations, a separate wiring owner is used. In the simplest case, a general wildcard rule permits any access, which causes an execution time increase of 19–22% compared to the baseline. The next configuration includes a simple rule that controls direct and indirect PIC access using a role-based subject template ("[*role* = 'Client'] **for** **") together with a type-based scope restriction. With a total overhead of 24–29%, the measurements show a relatively small decrease in performance. Many typical authorization constraints can already be expressed with similar rules.

In the "complex rule" configuration, the previous rule is extended by including a minimum age in the subject template and a scope selector that matches only half of the relevant entries according to their priority property. The additional checks themselves have almost no impact on performance, but for large entry counts, the scope restriction increasingly affects the evaluation time of the wiring guard, which leads to an overhead of 58% for 10,000 entries. In the "context-aware rule" setting, a condition is added that checks if a specific registration entry is included in the PIC with a property that

references the user ID via a dynamic parameter. This check increases the execution time by less than 11% for all entry counts, as the context entry can be quickly retrieved via its unique type.

As expected, the inclusion of a second, identical authorization rule increases execution times further, but only by 9–12%. The performance impact off adding 100 extra, non-applicable rules for the "full policy" configuration is only slightly higher (additional execution time increase of 11–14%).

These benchmarks demonstrate a reasonable authorization overhead also for the Java Secure Peer Space implementation. The relative overhead is higher compared to the MozartSpaces benchmarks, but this can be explained by the simple wiring processing logic and the missing need for serialization, which makes wiring execution faster than MozartSpaces operations on remote cores. The absolute overheads are similar to those measured in the MozartSpaces data query benchmarks (see Section 5.6.2), which also include two authorization checks (for writing into the request container and querying the target container). In both cases, the authorization time amounts to around 0.1 ms per iteration for a full policy configuration in a mostly empty container. However, the Java Secure Peer Space provides a significantly better scalability with the number of container entries, which can be explained by its more efficient scope evaluation that does not require computing the permission of all entries before the actual query.

6.6.3 Concurrent Request Benchmarks

Another benchmark series addresses the sending of requests to a target peer located at a server space. Wirings within client peers concurrently generate request entries and write them to the PIC of the server peer using the destination mechanism. As soon as the corresponding wiring in the server peer returns a response entry, the client peer sends its next request. For each benchmark round, 10,000 requests are processed in total, which are split upon the configured number of clients. For the sake of convenience, the client peers are all located in the same client space. Due to the middleware runtime architecture, they still operate independently using separate threads.

At the server space, write permissions to the PICs of the RTP and its nested server peer have to be granted to the clients (i.e., subjects of the form "*ClientUser* @ *ClientRuntime*"). The server wiring is owned by the local runtime user, thus internal take and write access is always allowed. As the focus of the benchmarks lies on the server performance, access control is not enforced by the client space runtime.

Figure 6.15 depicts the results for benchmarks with different security configurations and an increasing number of active client peers. All configurations show a good scalability with concurrently running client threads, which provide more than double throughput compared to a single client scenario, where the server idles while client-side processing takes place.

Compared to the wiring execution benchmarks, the overhead of the access control mechanism is clearly higher. For the general authorization scenario, which permits any access via wildcard rules on the server peer and its RTP, it is at least 77%. This can be mostly explained by the relatively high serialization overhead for the subject trees

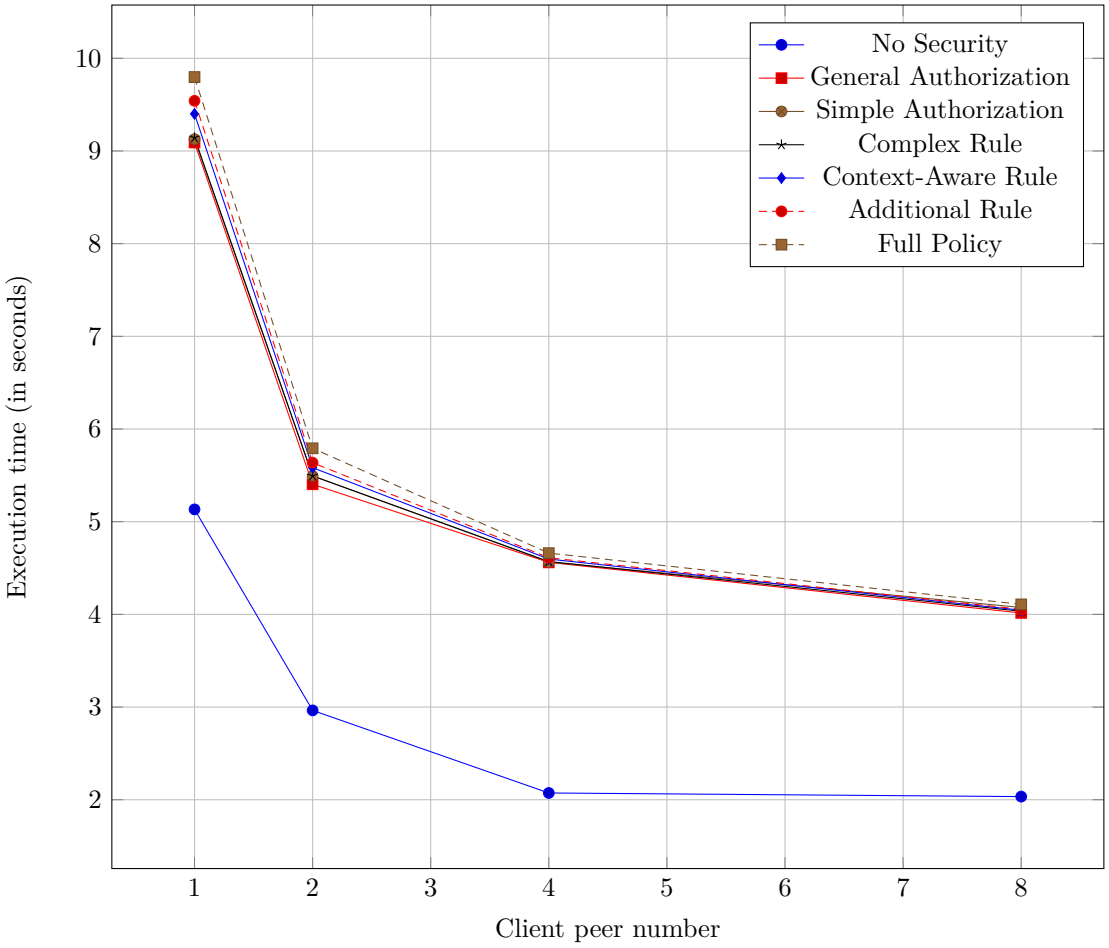

Figure 6.15: Concurrent request execution benchmarks with 10,000 iterations

that are attached to the request entries. Additionally, the Secure Peer Space runtime also automatically adds its own identity to each response entry. If the entries carried an actual application-specific payload (i.e., complex arguments and return values), the relative overhead would become less significant.

The "simple authorization" configuration includes a trust-based rule on the RTP, which allows all access by subjects from the client runtime (i.e., "* @ *ClientRuntime*"), and a simple privilege-based rule on the server peer that allows matching subjects ("[*role* = 'Client'] @ *") to write entries of type `Request`. In the "complex rule" setting, the second rule is again extended with an age check in the subject template and a priority-based scope selector. The context-aware rule adds the same condition as for the wiring execution benchmarks to the server peer rule, whereas the "additional rule" and "full policy" settings also retain their meaning. All these configurations cause less than 8% of additional execution time on top of the general authorization scenario.

The benchmarks show that policy complexity is not a major obstacle when authorizing remote requests. The limiting factor is mainly the size of the subject trees, which should therefore only include necessary security attributes. It has also been demonstrated that the access control mechanisms are well-suited for scenarios with concurrent requests. When assuming eight client peers, the absolute overhead amounts to around 0.2 ms per request, which is sufficient for most practical scenarios.

6.7 Critical Reflection

By providing a natural extension of the XVSM access control model and the Secure Service Space architecture, the presented approach enables the modeling of security for distributed workflows in open environments. Permissions for service invocations, data access, and administrative tasks are controlled via highly expressive and flexible authorization rules. Due to the decentralized policy definition, each stakeholder retains full control over access to its own components. The advanced subject concept enables rules based on the trustworthiness of the flow originator, delegated principals, and involved Peer Space runtimes.

The access control model provides multiple layers of protection, both locally via hierarchical policies and globally via trust-based rules. In a typical scenario, peers regulate who can access their functions via corresponding rules for each request type, while more specific rules in (possibly nested) sub-peers restrict how these functions operate for specific subjects and which kind of results they are able to return. Thus, coordination logic and related security constraints are encapsulated in each peer. Like in the Secure Service Space, service-centric and data-centric authorization can be combined, but corresponding rules are not limited to a strict two-tier architecture and may span multiple peers. Additionally, trust-based rules may be specified at the RTP level, which regulate which subjects are trusted to interact with the local Peer Space. In many cases, it is sufficient to check the identity of the authenticated runtime user, while matching other principals via wildcards. A chain of trust is established when each runtime user trusts its respective predecessor in the authentication chain to establish reasonable trust-based rules that prevent access from malicious sources. If other runtime users are not fully trusted, the whole subject tree may be evaluated instead. Thereby, the trustworthiness of involved principals may be assessed independently several times within a flow.

In order to ensure the practical feasibility of the Secure Peer Space, several issues have to be addressed. As demonstrated by the performed benchmarks, scalability is improved compared to XVSM. While access control has to be enforced at multiple points within the peer hierarchy, the individual authorization checks can be executed more efficiently. However, the tradeoff between the complexity of an authorization policy and its performance overhead still has to be considered. Alternative security configurations can be used to simplify policy management in specific scenarios. When only a single security domain is needed, hierarchical SPCs may be more difficult to manage than a centralized authorization policy. This issue can be resolved via a special system peer that manages the authorization policy of the whole space. An authorized administrator

can then send a high-level policy to this access control peer, which translates it into corresponding rule entries that are installed at different peers. Another optimization is the deactivation of authorization checks for certain sub-peers, which also reduces the performance overhead. In this case, access control is performed in the parent peer. As a special case, all rules may be included in the SPC of the RTP. However, this strategy negates the security advantages of the layered protection principle. Furthermore, if no rules for output restrictions are required, authorization on any POC can be simplified by implicitly permitting any write access by internal wirings.

The prototypical implementations have to be improved in order to support the complete coordination and access control functionality of the Secure Peer Space. The envisioned modeling tool should also incorporate the definition of authorization policies with accompanying sanity checks, thus simplifying security administration tasks. Further measures are necessary to increase the resilience of the middleware prototypes against attacks. This includes proper sandboxing for services, so that they cannot interfere with the runtime behavior or block indefinitely, and restrictions on the query mechanism, as the currently used Java predicates may execute arbitrary code.

CHAPTER 7

Secure Coordination Patterns

In order to provide examples for the proper utilization of the proposed access control approach and to demonstrate its feasibility, a selection of secure coordination patterns is presented in this chapter. Such patterns analyze common collaborative scenarios occurring in open distributed systems and suggest reasonable solutions for associated collaboration and security challenges. Existing coordination patterns focus solely on the necessary coordination among the involved components, whereas secure coordination patterns also incorporate access control in the form of appropriate authorization policies that prevent illegitimate interactions.

Secure coordination patterns are specified using a combination of textual descriptions and a corresponding *solution model*. Following the approach of similar pattern collections [GHJV95, CCF+00, SFH+06], a pattern specification is divided into several sections:

- A fitting **pattern name** and a short introduction that outlines its overall **intent**.
- A **problem** statement that describes the targeted scenario, the involved components and stakeholders, as well as related coordination and access control requirements.
- The suggested **solution**, which consists of a solution model and a corresponding explanation.
- An overview of possible pattern **variants** with modified requirements and associated adaptations of the solution.
- An analysis of expected **consequences** of the presented solution, which includes benefits and possible problems.
- An **applicability** section, which discusses practical scenarios where this pattern may be relevant.

Pattern solutions are specified using the Secure Peer Model, which provides more sophisticated coordination and access control features than the previously examined Secure Space. In principal, however, most of the examined patterns could also be addressed by XVSM-based solutions. In fact, the Secure Service Space from Section 5.4 can be seen as a secure coordination pattern on top of the Secure Space architecture.

A solution model contains a set of partially specified peers and wirings that are tightly related to each other, which constitutes the abstract coordination logic of the pattern. Additionally, the ownership of relevant entities is declared and corresponding authorization rules for the involved containers are defined. The generic character of pattern solutions is realized via *configurable properties*, which were introduced in [KCS15] to enable parametrization of peers and wirings. Any part of a peer or wiring specification can be made configurable. This includes source and target containers for links, service references, and PMQ fields, as well as property values used within PMQ selectors and link assignments. When integrating a pattern solution into an application, each of these customization points has to be bound to a concrete value. For secure coordination patterns, this concept can be extended so that it also applies to elements of authorization rules.

The pattern solution models described in this thesis are built upon basic mechanisms of the Secure Peer Model as described in Chapter 6: storing information and triggering flows by pushing entries to local or remote containers, processing entries with wirings using different access modes, retrieving data via dynamic wirings, and specifying authorization rules for all of these interactions. For the creation of the solutions, a few general design principles are considered. A solution model describes a rather generic view of its pattern to enable its application for different scenarios. Therefore, only relevant coordination logic and authorization constraints are included. In order to obey the principle of least privilege, only necessary interactions are permitted and rules are rather restrictive. In addition, a hierarchical policy structure is followed, where general, trust-based rules are defined at a higher level, while peer-specific restrictions are specified in the corresponding sub-peers. Therefore, authorization policies are able to target multiple security layers together. At the communication layer, they indicate which messages are trustworthy, while at the service and data layers, they define the actual access privileges on internal resources.

Secure coordination patterns exist at different abstraction levels. Basic patterns (Section 7.1) add suitable authorization constraints to simple coordination tasks that are part of many interaction protocols. Section 7.2 shows how more complex patterns can be composed from basic ones. The selected pattern examples provide generic design recommendations that can be applied to a wide range of different scenarios. To confirm their validity, the presented pattern solutions have been implemented and tested using the Java Peer Space. The long-term vision is to provide an extensive pattern catalog for different collaborative scenarios, which would enable a pattern-based design methodology where secure distributed applications could be mostly composed of configurable building blocks with proven qualities, only requiring a minimal amount of additional glue code. This new software engineering process is outlined in Section 7.3.

7.1 Basic Patterns

This section describes several simple patterns that address common coordination strategies in a generic way, while suggesting appropriate authorization policies for each involved stakeholder. They have been abstracted based on observed and approved design decisions related to the modeling of distributed interactions for the examined application scenarios. The corresponding pattern solutions define two things: how the required coordination logic can be expressed with the Peer Model, and which authorization rules are necessary to protect the modeled interactions.

For the specification of the pattern solution models, the graphical Peer Model notation is adapted. Peers are connected with associated rule specifications as well as the ID of the owner. Configurable properties are realized by means of *pattern parameters*, which are denoted with a "$" sign and can be statically assigned to different values depending on the corresponding application scenario. To distinguish them from dynamic parameters within authorization rules and variables within link specifications, which are both resolved at run time, pattern parameters are written between parentheses and in upper case. A parameter's name should indicate its meaning in the context of the pattern. For instance, an entry type within a guard PMQ may be set to $\$(REQUEST)$, while a configurable subject template within an authorization rule may be denoted as $\$(CLIENT)$.

For a more comprehensive view, also the dynamic behavior of the solution is shown when it is relevant for the pattern. Entries and their relevant properties are visualized if they are injected via the API or by an unspecified wiring. Dynamically created peers, wirings, and rules are also included in the model and highlighted with a red border[1]. For properties that may be different for each invocation, an additional variable type termed *dynamic pattern placeholder* is introduced. Such placeholders are syntactically similar to pattern parameters, but use a "#" sign instead of "$". They provide a shortcut for showing the correlation between different elements of a pattern solution. Their value is determined dynamically based on certain input entry properties. For correlation between static and dynamic parts, a dynamic pattern placeholder may be bound to a specific value by adding an expression of the form "$[= \#(NAME)]$" to a link assignment statement of a static wiring in the extended graphical notation.

In order to keep the presented solution models simple, several assumptions and simplifications are made:

- All involved users share the same domain and identity provider. Thus, they can be uniquely identified via their user ID.
- The matching mechanism for principals within a subject template (by ID, role, attribute condition, etc.) is left unspecified by using configurable properties. Thus, an authorization rule may apply to all authenticated subjects, a specific group, or an individual user.

[1] It would be possible to define the complete semantics using a static view of the model, but then every dynamically created element would have to be specified fully within the emitting wiring, i.e., by setting each required property on the corresponding action link. Due to the complexity of some meta entries, such an approach is clearly not feasible for the graphical notation.

- Service implementations are not explicitly specified. Instead, they are abstracted via configurable properties. The relevant coordination logic is fully defined via the Peer Model's wiring notation.
- If not stated otherwise, wirings and sub-peers are owned by the corresponding peer owner.
- In order to keep the resulting subject trees simple, the involved principals are assumed to be the runtime users of their respective Peer Spaces.
- Runtime peers and their authorization rules are omitted if they are not relevant for the respective pattern. In this case, general trust-based rules are assumed that state which kind of subjects may interact with the local Peer Space.
- To improve the clarity of the graphical notation, data and context entries are managed in PICs instead of using separate peer containers.
- Meta containers are only shown when they are relevant.
- To increase readability of the models, names of peers, wirings, rules, principals, and custom coordination properties are not defined as pattern parameters, although they may be modified when applying the pattern.
- It is assumed that configurable addresses within `DEST` properties refer to the corresponding communication partner. In a similar way, pattern parameters within subject templates have to match the respective principals.
- Exception handling is omitted in the modeled solutions.

7.1.1 Stateless Service Invocation

A stakeholder offers stateless services that shall be accessible for remote users via request-response communication.

Problem: A server provides different services with well-defined functionality. Clients can invoke them by sending service-specific requests with associated parameters. The service then computes a corresponding result and returns it to the original invoker. The server should be able to handle concurrent access by multiple clients, which do not have to be known in advance. The services are assumed to be stateless, i.e., no state information about the interaction with any client has to be stored and the service result solely depends on the data within the request. At the server side, it should be possible to specify authorized clients for each offered service. On the other hand, clients have to be protected from spoofed results and unsolicited messages that strain their resources.

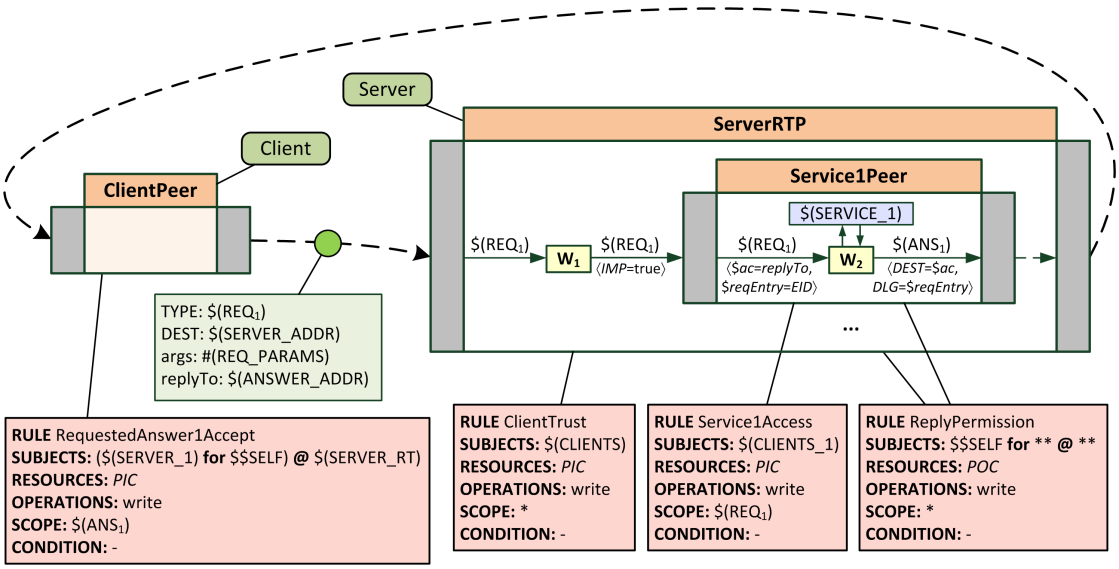

Figure 7.1: Solution model for Stateless Service Invocation pattern

Solution: A suitable solution model is shown in Figure 7.1. The flow is started by a request entry (with configurable type $\texttt{REQ}_1$) owned by a specific client user, which is targeted at the PIC of a server peer (with address `SERVER_ADDR`) owned by a corresponding server user. Additionally, the entry contains two custom coordination properties for the service-specific request parameters (within the nested `args` property) and the answer address (`replyTo`), which points to a peer controlled by the client. In the shown scenario, this is the same peer that has emitted the request, but an explicit answer peer would be required if the request entry had been injected by the client user via the API. As the request creation and response handling processes are not relevant for the core objective of this pattern, they are not specified here.

At the server, services are encapsulated in sub-peers of the RTP. A service peer is explicitly linked to its associated request type $\texttt{REQ}_x$ by a wiring that moves entries with the corresponding type from the RTP's PIC to its own PIC. The incoming request entry then triggers a wiring in the service peer, which may cause arbitrary computations based on the request parameters. Using a local variable (`ac`), the destination of the emitted answer entry ($\texttt{ANS}_x$) is set to the answer container address specified in the request entry.

Access control is governed via multiple types of rules. For the PIC of the server's RTP, a general trust-based rule is used to specify which clients are allowed to interact with the server (indicated by the `CLIENTS` parameter). For each service, the server user can then grant write permissions on the service peer's PIC to a more restrictive set of subjects (matched by `CLIENTS_`X) for the respective request entry type (set in the scope). The configurable subject template parameters denote the set of authorized clients, e.g., based on their role properties. As the server user is equivalent to the local runtime user, it can impersonate the accessing subject (i.e., "*Client*" in the example) for the invocation of the service peer. Wiring W_1 selects impersonated access mode for

forwarding the received entry by setting the `IMP` flag. Thus, the subject property of the input entry remains unchanged.

Take operations by the wiring guards are implicitly permitted, but actions related to the creation of answer entries are subject to access control because they use indirect access mode. Therefore, additional rules have to be specified for the service peers and the RTP, which allow all write operations on the POC by the peer owner (denoted by `$$SELF`) on behalf of arbitrary subjects. Wiring W_2 sets the delegator by linking the `DLG` property of the emitted entry to the entry ID of the input entry with the help of a local variable (`reqEntry`). Thus, the subject for the emission of answer entries corresponds to "*RuntimeUser* **for** *Client*" in the example.

When such an entry arrives at the client peer, its subject is authenticated as "(*Server* **for** *Client*) **@** *Server*". Corresponding rules ensure that only responses with matching subject and entry type are accepted. In the `subjects` field, the `SERVER_RT` property shall match any trusted server that may process corresponding requests for the client, whereas `SERVER_1` represents the principal template for the concrete service owner, which also corresponds to the server runtime user in the example. If the client does not care about this service user, a wildcard may be used for that parameter. As a protection for misrouted responses, only entries with the client peer owner as originator are considered.

Variants:

- **Fire-and-forget:** Not every service may produce a response. In this case, only PIC rules at the server side are required.
- **Remote answer containers:** The answer entry may also be sent to a third-party peer, from where the client can pick it up at a later time. Rules at this peer have to ensure that clients may only access answers to their own responses via context-aware scope queries that compare the accessing subject with the originator of the service invocation flow (cf. owner-based access in Section 7.1.3).
- **Parameter-dependent access:** More fine-grained service access rules may be specified by enriching the scope with a selector that restricts the parameter space for certain subjects, e.g., "FactorizationRequest $[\![args.val < 1\,000\,000]\!]$".

Consequences: From a coordination perspective, this pattern decouples clients and servers to some extent. Servers need not be aware of specific clients before handling their requests, and clients do not depend on the internal sub-peers and wirings of the server, but only have to know the server address and a previously defined request entry structure. Forwarding of requests via configurable server wirings instead of directly addressing sub-peers also enables transparent server-side switches between different service implementations without affecting the client logic.

For the server, the combination of a general RTP rule for overall access with fine-grained, service-specific rules provides a dual protection layer. The client-side rule

prevents attackers from sending fake results to the client, either directly or by using a spoofed answer address within a request sent to a legitimate server. However, the client still has to trust the server to provide a valid subject tree within its answer entry. It is also not ensured that the response is actually linked to a currently active request. For more restrictive access to the client, additional mechanisms are required (cf. Section 7.1.4).

Privacy is not protected by such a policy, as servers may freely choose to forward requests and responses to third parties. If privacy is relevant, a mobile agent scenario should be favored, where service peers are moved to Peer Spaces of their respective clients in order to ensure a controlled execution environment. In this case, outgoing communication may be governed via rules on the POC of the client's RTP (cf. information flow control in Section 7.1.7).

Applicability: This pattern is the foundation for any distributed workflow and can be used whenever a peer needs to invoke the well-defined functionality of another peer owned by a different stakeholder. It can be seen as a Peer Model equivalent to common technologies like RPC and Web services. However, also flexible P2P architectures are possible where peers alternately take the role of a client or server, respectively.

Like in the Secure Service Space pattern from Section 5.4, a service-centric form of authorization is supported. For controlling indirect access to data or other services on behalf of the client, a combination with additional patterns like Proxy (Section 7.1.2) and Shared Data Storage (Section 7.1.3) is required.

7.1.2 Proxy

Certain resources may only be indirectly accessed by regular users via a proxy.

Problem: Servers provide some kind of functionality, but clients must not access them directly. Instead, access is protected by a dedicated proxy, which invokes the servers on behalf of the clients. This proxy may provide application logic of its own and transform the client command accordingly before forwarding it to one or more servers, which could be selected based on information provided by the client, a stored internal state, or a proxy-specific policy (e.g., round-robin or random scheduling). From the perspective of the client, the proxy acts as the server, as any subsequent delegations happen in a transparent way. However, the server is still aware of the original client and may restrict indirect access based on its identity. Thus, in order to invoke functions or cause state changes at the server, clients need to be authorized by the intermediate proxy as well as by the actual servers.

Solution: Figure 7.2 shows a scenario where a command entry is relayed to a single target server via a proxy peer. Wiring W_1 takes such entries and processes them in its service, which sets the `target` property to the address of a remote peer that is suited for executing the requested command. The target choice may be configured statically at the proxy or selected dynamically using an unspecified decision mechanism. The service

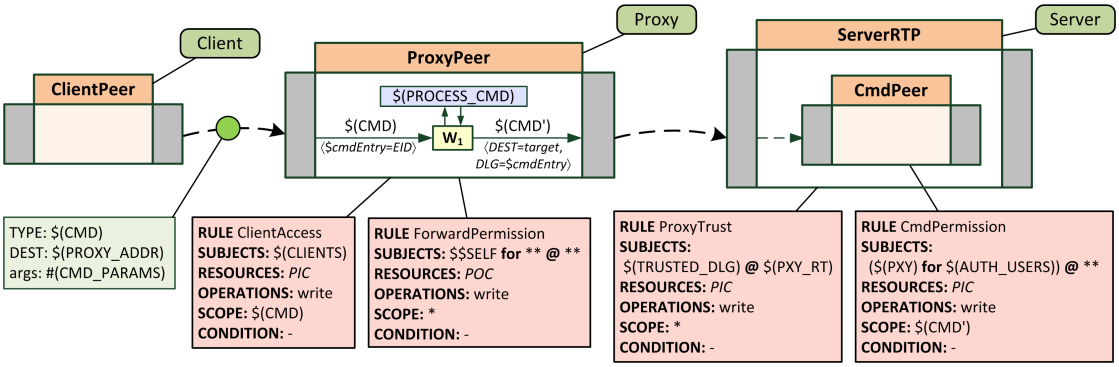

Figure 7.2: Solution model for Proxy pattern

may also add additional parameters or modify existing ones, thus changing the parameter for the entry type from `CMD` to `CMD'`. On the action link, the destination is set to the target[2] and indirect access on behalf of the input entry owner is configured via the `DLG` property.

Access to the proxy and thus to the functionality provided by the command is controlled via a rule on the proxy's PIC that restricts access for the specific `CMD` type to authorized subjects matched by the `CLIENTS` parameter. As for the Stateless Service Invocation pattern (Section 7.1.1), an additional POC rule is required to enable the emission of entries using indirect access. Each server that is managed by the proxy may then specify its own authorization policy based on its trust in the proxy and the delegating clients. This policy can be split into two parts. A trust-based rule for the RTP's PIC shows that communication with the server is only possible via a Peer Space provided by a valid proxy runtime user (matched by `PXY_RT`). The `TRUSTED_DLG` parameter corresponds to a subject template that specifies which kind of delegations are allowed via the proxy. If full trust is assumed, this may correspond to an unrestricted subject (i.e., "** @ **"). In the provided solution model, the subject "(*Proxy* **for** *Client*) **@** *Proxy*" has to be matched by this rule. The second rule type defines actual permissions for writing the forwarded command type to the targeted command peer's PIC. It defines which clients (`AUTH_USERS`) may indirectly invoke the offered command via which proxy (`PXY`). For the latter parameter, a wildcard may be inserted if the server does not need to distinguish between different proxy users. As the destination mechanism ensures that any incoming entry passes the RTP's PIC (and thus has to satisfy the trust-based rule) before entering a sub-peer, the authentication chains do not have to be checked again.

Variants:

- **Request-response:** Like in the Stateless Service Invocation pattern (Section 7.1.1), a command may also cause a response that needs to be routed back to the client.

[2]The `DEST` property could also be directly set by the service, but it is recommended to explicitly include properties relevant for coordination in the modeled design in order to provide a clear separation between application and coordination code.

The server may directly send back an answer entry using a specified answer address, but then the client needs to include an access rule that matches the server user. Alternatively, the responses may be returned via the proxy, which forwards them to the originator. In this case, the indirect write operations by the server to the proxy and by the proxy to the client must be permitted.

- **Chained delegation:** Multiple levels of indirection may be used by chaining instances of this pattern together. Instead of a server, the target of a proxy may be another proxy, for which the first proxy acts as the client. As indirect access is used for every step, the subject tree is extended iteratively. A chain of trust can be established, where every proxy establishes trust-based rules for allowing communication by the previous proxy, whereas the final target servers can still specify permissions based on the flow originator.

- **Multiple servers:** The proxy may replicate a command or split it into several sub-commands. In this case, the corresponding wiring emits multiple entries that are sent to different servers. The access control rules for the proxy peer and each server peer remain the same.

Consequences: Using this pattern, a further decoupling between clients and servers is achieved, as they only communicate indirectly via the proxy. The proxy peer transparently provides functionality of one or more servers and ensures a controlled invocation of their behavior.

Server security is increased, as direct access is only allowed by trusted proxies, which act as gatekeepers for client requests and ensure that only valid commands arrive at the servers. However, servers may still perform authorization based on the originating user. Thus, a double protection layer is realized, where the proxy performs a preselection of authorized users but the respective server controls the final access decision. The separation of trust- and privilege-based rules at the server peer simplifies policy definition, as the proxy user may be omitted at the sub-peer level.

Due to the inherent decoupling of this pattern, clients cannot control which servers are actually invoked. In order to prevent misuse of their provided information, they should therefore only send entries to trusted proxies.

Applicability: This pattern can be applied whenever server resources shall not be directly accessible by end users but only indirectly via trusted components. It can be compared with a reverse proxy approach in Web architectures, where a proxy server transparently accesses associated servers on behalf of a user. Similar to the gateway architecture of TuCSoN, nested protection domains can be built in this way. Another use case is load balancing, where the proxy peer dispatches requests to one of several server peers.

Proxy peers may also provide more complex internal logic and dynamically derive command entries and corresponding targets based on incoming client requests and an internal state. Thus, management of distributed peers may be realized using the

Proxy pattern. In the firewall management scenario, the SMC acts as a proxy between administrators and managed firewalls. Another example would be a replication manager component that controls access to distributed replicas [CHKS14], which must not be directly manipulated by users to ensure consistency. In some cases, it may be sufficient if only the proxy peer enforces client-based authorization, but in order to preserve their autonomy, servers should at least have the possibility to specify additional constraints. Distributed firewalls, for instance, may only accept commands from the SMC when they were initiated by valid administrators from their own organization.

The described approach is often combined with other patterns, like Stateless Service Invocation (Section 7.1.1) and Shared Data Storage (Section 7.1.3), to model indirect access within more complex interactions.

7.1.3 Shared Data Storage

Data-driven communication among two or more stakeholders shall be enabled in a decoupled and secure way.

Problem: Distributed components need to exchange data or events in an ad-hoc way without having to know each other. Thus, a third-party storage server is required that provides a space-like coordination abstraction with write, read, and take operations. Clients should be able to retrieve information using expressive queries that may block if the corresponding data is not yet available.

For this, a flexible access control mechanism is required that specifies which users may write, read, or delete which data. Certain information may be shared publicly, while it shall also be possible to use private storage areas for managing the state of a specific user or for communication within closed groups. Queries shall be performed in a transparent way, i.e., only accessible data may be visible for each client. Additionally, requestors have to be protected from faked query results.

Solution: Figure 7.3 shows how XVSM-like functionality can be bootstrapped with the Peer Model. The storage peer, which does not have any internal logic in the form of predefined wirings, holds data entries in its PIC (or in a separate state container). Clients may store entries by using this remote container as destination, and they can retrieve the data again by installing dynamic wirings on the storage peer via its WSC. In the example, the same client writes and reads entries, but in general, different peers may perform these operations.

The wiring specification corresponds to a read or take request in XVSM. It defines the query via the PMQ of its guard, represented by a configurable entry type (`DATA`) as well as the dynamic pattern placeholders `CNT` and `SEL`. When the query is fulfilled, the wiring's action sends all retrieved entries to a configurable answer destination (`C_ADDR`) corresponding to the requesting client. Due to the set `repeat` property, the wiring triggers at most once and then uninstalls automatically. Blocking behavior with timeouts

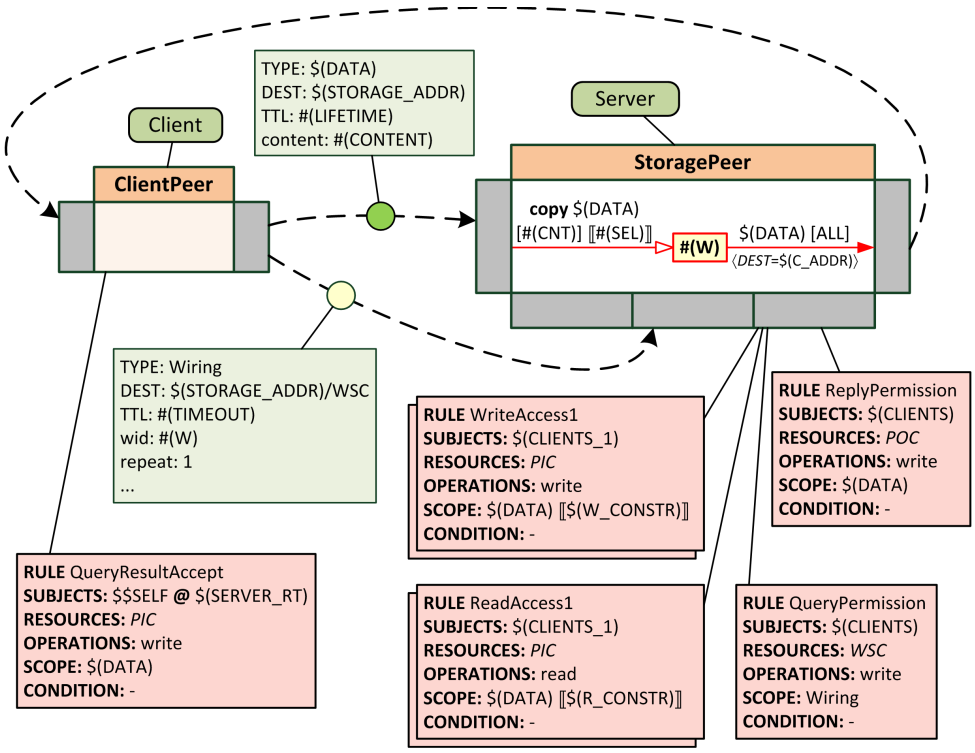

Figure 7.3: Solution model for Shared Data Storage pattern

can be achieved via the wiring's `TTL` property. Similarly, also the lifetime of written entries can be limited.

Fine-grained access control on the storage peer is enabled via rules that define permitted entries based on possibly complex scope restrictions for each relevant user group matched by a given subject template. The depicted solution model includes rules that allow selected clients (matched by the `CLIENTS_X` parameter) to write and read entries with type `DATA` and specific properties as defined by the included constraint parameters (`W_CONSTR` and `R_CONSTR`). Similar rules may also be defined for deleting entries using the take operation. Using this approach, separate partitions for different groups can be established, e.g., when the pattern parameters within the scope are replaced by "Event $[\![group = \text{'SBC'}]\!]$" so that subjects affected by this rule may only write or retrieve event entries with the right `group` property.

For installing dynamic wirings, all clients (matched by `CLIENTS`) are granted write permissions on the WSC. The access operations of these wirings are authorized based on their owner, i.e., the querying client. Besides the read or take permissions on the PIC, they also require a write permission on the POC to enable a response. This response has to be accepted by the client via a rule on its PIC, which only allows data entries from one of its own wirings installed at a trusted storage server runtime (matched by `SERVER_RT`). In the example, the relevant subject tree is "*Client* @ *Server*".

Variants:

- **Indirect access:** Instead of allowing direct installation of wirings by the client, the storage peer provides a predefined wiring that accepts request entries for reading, taking, and also updating data. This wiring then installs the required dynamic wiring on behalf of the original user. Permissions for writing the request entries must be added and subject templates in the authorization rules have to be modified to reflect the indirect access.

- **Owner-based access:** By exploiting the dynamic parameter feature of the access control model, separate partitions for different subjects can be established automatically without the need for adding different rules. Instead, for each query operation, relevant security attributes of the client (or the flow originator in case of indirect access) are compared with respective properties of the entry owner within the configurable `R_CONSTR` property. The selector "$originator.userId = \$originator.userId$" ensures that each client can only access its own entries, while "$originator.dept = \$originator.dept$" supports data exchange within the own department.

Consequences: This pattern enables a space-like abstraction for decoupled interaction among peers. In contrast to XVSM, query operations have to be modeled explicitly via dynamic wirings, which increases complexity, but also allows for flexible modifications of their semantics. In the presented solution, only the basic query behavior is specified. For instance, empty results (e.g., with count `ALL`) and exceptions (e.g., due to operation timeouts) are not detected by the client. If this is necessary, the pattern has to be extended correspondingly.

On the storage peer, fine-grained access control is supported based on user attributes and data properties, whereas denied entries are automatically hidden. The client rule prevents forging of query results by other clients, although trust in the storage peer is still required. Manipulation of wirings by remote clients has to be handled with care, as such wirings may send entries to arbitrary destinations. Receivers may be tempted to accept such entries when they come from a trusted Peer Space runtime. It is possible to mitigate that risk with restrictive rules on WSC or POC (e.g., in the shown solution model, wirings may only emit data entries), but other peers should still not assume the trustworthiness of a subject solely on the basis of its authenticating runtime, especially if the actual invoker within the delegation chain is not familiar. Thus, it is recommended to either use separate runtime users with limited privileges for providing storage functionality, or follow the indirect access variant instead.

Applicability: Like the previous patterns, this pattern represents a basic building block that may be used for many distributed application scenarios. It enables ad-hoc communication of decoupled components via a shared space, e.g., collaborating robots in a home automation scenario, as described in [CJK15]. Storage peers can also act as simple databases that manage relevant state information. The flexible authorization

policy supports public data clouds (with possible content-based restrictions) as well as private data stores for specific groups.

In order to realize space-based services like in the Secure Service Space architecture, this pattern may be combined with the Stateless Service Invocation pattern (Section 7.1.1). For computing its result, a service may query local or remote data on behalf of the client using indirect access. In this case, the owner-based access variant enables the easy management of user-specific state information, as only entries associated with the originating principal are visible for the respective dynamic wiring. Owner-based access can also be applied for restricting access to the meta model. Administrative rules may grant certain users access to WSC, PSC, and/or SPC, but they may only read and delete their own meta entries.

7.1.4 Dynamic Response Handling

Clients shall be able to dynamically specify asynchronous response handling logic for any of their requests.

Problem: Triggered by specific events, clients invoke server functionality in an asynchronous way. Each client may send multiple requests concurrently to the server and the corresponding responses have to be handled accordingly by the client. Response handlers are specified individually for each request, as their behavior may depend on the initiating event. Timeouts may limit the validity of requests. Each handler must only be invoked once, namely by its awaited response. Therefore, clients should only accept responses for their own, active requests.

Solution: Figure 7.4 shows a solution for this problem that uses dynamic wirings as response handlers. The flow is triggered at the client peer by a specific event or start token (`START`), which includes a unique `id` property. The configurable service of W_1 then generates a corresponding request (`REQ`) that is sent to the server peer address. The request ID (`reqId`) is set to the received ID via a local variable, whereas the configurable timeout parameter specifies its expiration time. The time-to-live mechanism ensures that the entry will vanish at the server if it is not processed by then.

Additionally, the wiring emits rule and wiring specification entries into the respective meta containers. The rule permits the server to transmit the expected response entry (`RESP`), whereas the dynamic wiring provides the response handling logic. For easier request correlation, all three output entries include the request ID within their corresponding ID properties. The dynamic pattern placeholders `R`, `W`, and `ID` are used to reference these values outside of the wiring's scope. Furthermore, the request timeout parameter is reused as `TTL` value for both meta entries, which ensures that responses for expired requests are ignored.

The internal server logic is not relevant for this pattern. It is only assumed that the server peer accepts requests from the client (indicated by the `ClientAccess` rule) and eventually returns a response entry that can be correlated to the original request via its

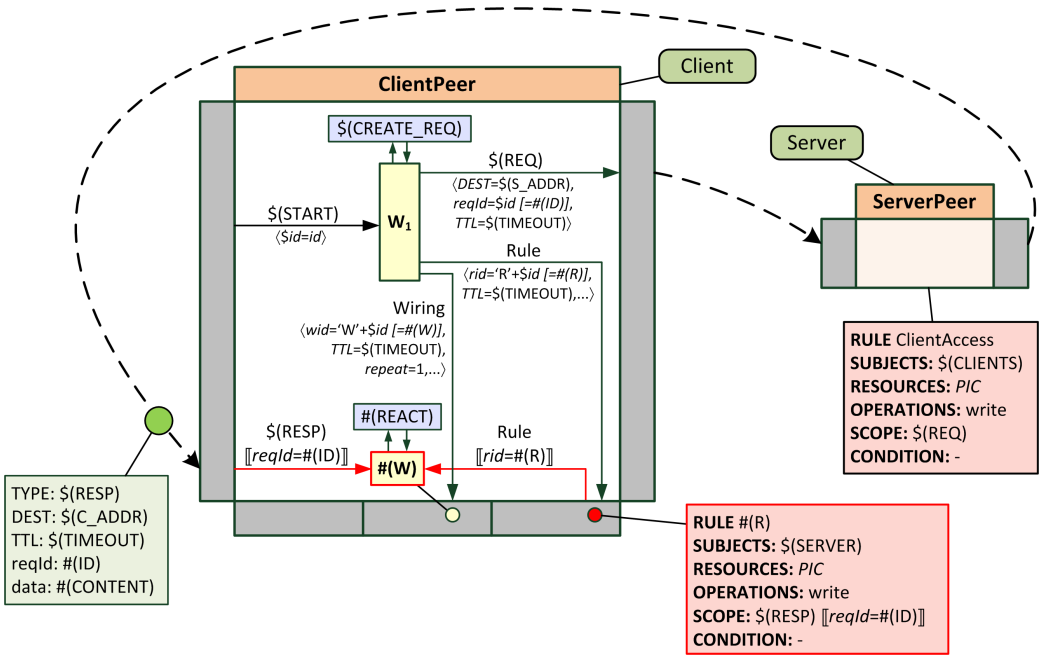

Figure 7.4: Solution model for Dynamic Response Handling pattern

`reqId` property. The origin of the `START` entry is also not modeled. If its owner were different from the client, an additional authorization rule would be required.

The dynamically set rule at the client peer grants the configured server the permission to write a response entry to the PIC, but only for the specified request ID. The corresponding subject template (`SERVER`) may match a specific server or a compound subject if indirect access mode is used. The dynamic wiring takes the response entry with the specified ID and processes it within its service, which may be selected by W_1 depending on the generated request. Its second guard removes the associated rule entry from the SPC based on its ID, thus revoking the corresponding permission after the response has been handled. Due to the set `repeat` property, also the wiring itself is automatically removed after one execution. Using this approach, multiple requests with corresponding wirings and rules may be active at the same time without affecting each other.

Variants:

- **Flow-based correlation:** The integrated flow correlation mechanism of the Peer Model may be used to simplify the coordination logic. Related requests, responses, and meta entries share the same flow ID. The response handling wiring does not require the specification of a concrete ID value, as the wiring semantics automatically combines each response entry with the corresponding rule entry[3]. As the wiring

[3]This assumes that no rule entry without a flow ID is present in the SPC, as these would also be viable for consumption by the guard link. Otherwise, the selector would have to filter dynamic response handling rules, e.g., based on a special flag property.

does not depend on a fixed ID anymore, a single, predefined request handling wiring may be sufficient as long as the service logic does not require adaptations based on the given request. For the rule definition, however, the flow ID still has to be specified within the scope selector. Timeouts are enforced via the rule's `TTL` property.

- **Response stream:** One request may also cause multiple response entries. If their number is fixed, specific rule and wiring specification entries may be created for each of them. Otherwise, the dynamic wiring's `repeat` property is set to infinite and its guard link from the SPC is removed so that response handling remains active until the defined time-to-live expires.

Consequences: This pattern enables concurrent requests with an automatic correlation of responses, which are handled independently from each other. The timeout mechanism ensures that outdated requests are not processed.

The dynamic rule specification enables strict policies according to the principle of least privilege. Each rule only allows a response by the invoked server for a concrete request. Permissions are automatically revoked when the response has been processed or the timeout has been reached. Race conditions may occur when multiple responses for the same request are concurrently sent or a response is received right before the wiring expires, because the dynamic wiring may not trigger immediately. However, in any case at most one response is actually processed and any remaining response entries will eventually expire due to their own timeouts.

Applicability: This pattern can be combined with other patterns like Stateless Service Invocation (Section 7.1.1) and Shared Data Storage (Section 7.1.3) in order to add support for asynchronous requests or queries, as well as stricter client-side permissions. It resembles the registration of a callback function for each request that appears transparent to the server. In a generalized form, it can be applied to any situation where ad-hoc permissions are required when a peer is waiting for a specific entry.

7.1.5 Context-Based Access

Access to certain functionality of a stakeholder shall be determined by its configurable application state.

Problem: A server provides functionality to clients, but it may only be invoked while it is in a specific application state. This state can be modified by an external source, e.g., a remote administrator. The system must define context-aware permissions for clients and allow state changes by privileged users.

Solution: Figure 7.5 shows a scenario involving server, client, and admin peers. The client sends command entries (`CMD`) to the server, which are processed in some way

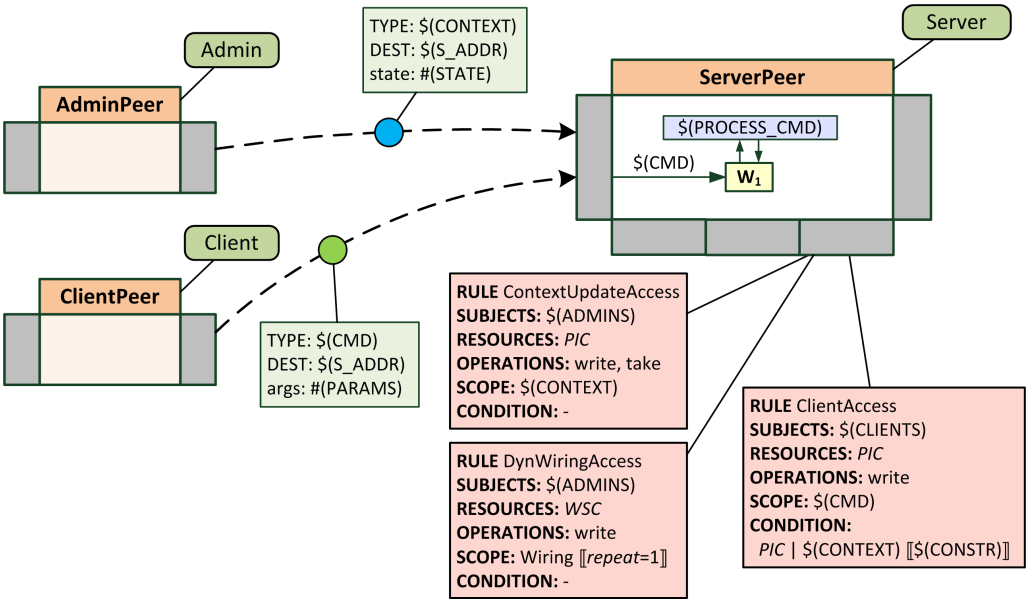

Figure 7.5: Solution model for Context-Based Access pattern

by wiring W_1. The application state is depicted via a context entry (`CONTEXT`) in the server's PIC (or alternatively in a distinct context container), which is set by the admin peer and includes context information within its properties. This peer may also update an existing context entry using a dynamic wiring, which is omitted in the figure. Entries from the client and admin peers may be sent by corresponding users via the API or predefined wirings.

The `ContextUpdateAccess` rule ensures that privileged administrators (matched by the `ADMINS` subject template) can update context entries, which in addition requires permissions to install dynamic wirings as granted by the `DynWiringAccess` rule. The `ClientAccess` rule provides a context-aware permission for the set of valid clients (covered by the `CLIENTS` parameter) to write command entries. Its condition checks that a context entry exists within the PIC and that it fulfills a specific constraint defined by the `CONSTR` parameter. For instance, a selector of "$state.phase \geq 2$" would only allow client access after the second application phase has been started.

Variants:

- **Registration-based access:** Besides the general application state, also user-specific context information may be stored in separate entries. Such user registry entries can be updated manually by an administrator or indirectly by the user through triggering a server-side registration wiring. By enriching the condition with a dynamic parameter that refers to the client's subject property, permissions may be restricted based on the stored user context. A possible condition would be "PIC | Registry $[\![user = \$invoker.userId]\!]$", which only enables access if a matching registry

entry with a `user` property corresponding to the invoking client user has previously been set. In case of self-registration, an explicit `user` property is not required, as the ID of the registered user is already included in the subject property of the registry entry and can be referenced in the rule instead. Using logical conjunction, such a condition may also be extended with additional constraints regarding other properties within the user's registry entry.

- **Internal context update:** Context entries can also be managed by the server itself without intervention by an external administrator, e.g., by means of internal wirings owned by the *Server* user. In this case, the authorization rules for the *Admin* user can be dropped.

Consequences: This pattern facilitates controlled invocation of server functionality, where permissions are dynamically unlocked based on context information provided by privileged users. The suggested approach simplifies access control, as administrators only have to change a simple context entry instead of constantly having to add or remove rules, which may lead to configuration errors.

In the presented solution model, it is assumed that only a single context entry (or a single registry entry per user) is present in the server's PIC. Otherwise, contradicting context entries may lead to undesired behavior. If the administrator is not fully trusted or there are multiple administrators that may interfere with each other, a context-aware rule could also be used to control write operations involving context entries, by including a condition that ensures that no duplicates exist, e.g., "`NOT` *PIC* | Context".

In some cases, context information may be relevant for multiple peers within the same runtime peer. As condition queries are limited to the local peer due to the encapsulation feature of the Secure Peer Space architecture, context entries and corresponding authorization rules have to be brought together in the same peer. This can be achieved via a synchronization mechanism for context entries or by performing authorization checks at a higher level within the peer hierarchy, i.e., in a shared parent of all context-dependent sub-peers.

Applicability: By incorporating configurable context information into the access decision, the pattern enables a coordinated execution of behavior among distributed peers, which is comparable to active access control as used by several workflow models like T–RBAC. Permissions are activated as long as the corresponding task, which may be represented via the context entry, is active. The approach supports the management of different application phases as well as the simple activation and deactivation of access permissions based on a single entry (e.g., as an emergency shutdown switch). Besides manual context changes by an external administrator, also internally managed context entries are possible, e.g., for blocking access when a certain workload threshold is exceeded. The registration-based variant provides functionality similar to typical Web application architectures, where users have to register to a platform and provide specific profile information in order to access a service.

The pattern is particularly suited for workflows that involve human interactions. In [CDJK12], a secure paper reviewing system is modeled with the Secure Space. Access to specific containers is only permitted when the workflow is in the right phase (e.g., "voting"), whereas additional registration-based constraints may apply (e.g., reviewers have to be registered as program committee members for a conference). In the firewall management use case, which was modeled with the Secure Service Space in [CDJ+13], only registered firewalls are able to send information to the SMC. Both models can easily be transformed to the Secure Peer Space architecture.

The Context-Based Access pattern may also be combined with other patterns like Stateless Service Invocation (Section 7.1.1) or Proxy (Section 7.1.2) in order to add context-based restrictions at the server.

7.1.6 Stateful Interaction

Two or more components need to interact using a multi-step coordination protocol that requires participants to retain information about the current state of the interaction.

Problem: Distributed components coordinate themselves using a well-defined protocol that consists of several sequential steps involving local processing and the exchange of corresponding messages. As participants may be involved in multiple of these protocol phases, they need to store state information that influences their actions in later phases. Within an instance of such a workflow, each step depends on a preceding message received from a remote partner, as well as the local state. Concurrent workflow instances shall be possible, which must not interfere with each other.

The message order defined by the protocol has to be enforced. Authorized users may initialize a workflow instance, but subsequent protocol steps shall only be possible when the distributed application is in the right phase and expects a corresponding protocol message from the respective source.

Solution: A solution model for a scenario with two interacting peers is shown in Figure 7.6, but the approach may also be generalized to protocols involving multiple peers. A flow is triggered by a start token at *PeerA*. Wiring W_1 initializes a local state entry in the PIC (or in a separate state container) and emits a message entry (with type $\texttt{MSG}_1$) that is targeted at *PeerB*. Both entries share the same flow ID, which corresponds to the flow of its input entry and is assigned using a local variable. Besides other state information defined by the configurable service, the state entry also has a `phase` property, which describes the current protocol phase from the viewpoint of *PeerA*.

The $\texttt{MSG}_1$ entry acts as start token for *PeerB*, which initializes its local state in the same way and returns a follow-up message with type $\texttt{MSG}_2$ to *PeerA*. Subsequently, several messages may be sent back and forth, whereas the corresponding wirings use the received message and the local state as input and provide a resulting message and an updated state as output (according to their service logic). For each interaction step, also the protocol phase is updated in the state entry. The automatic flow correlation

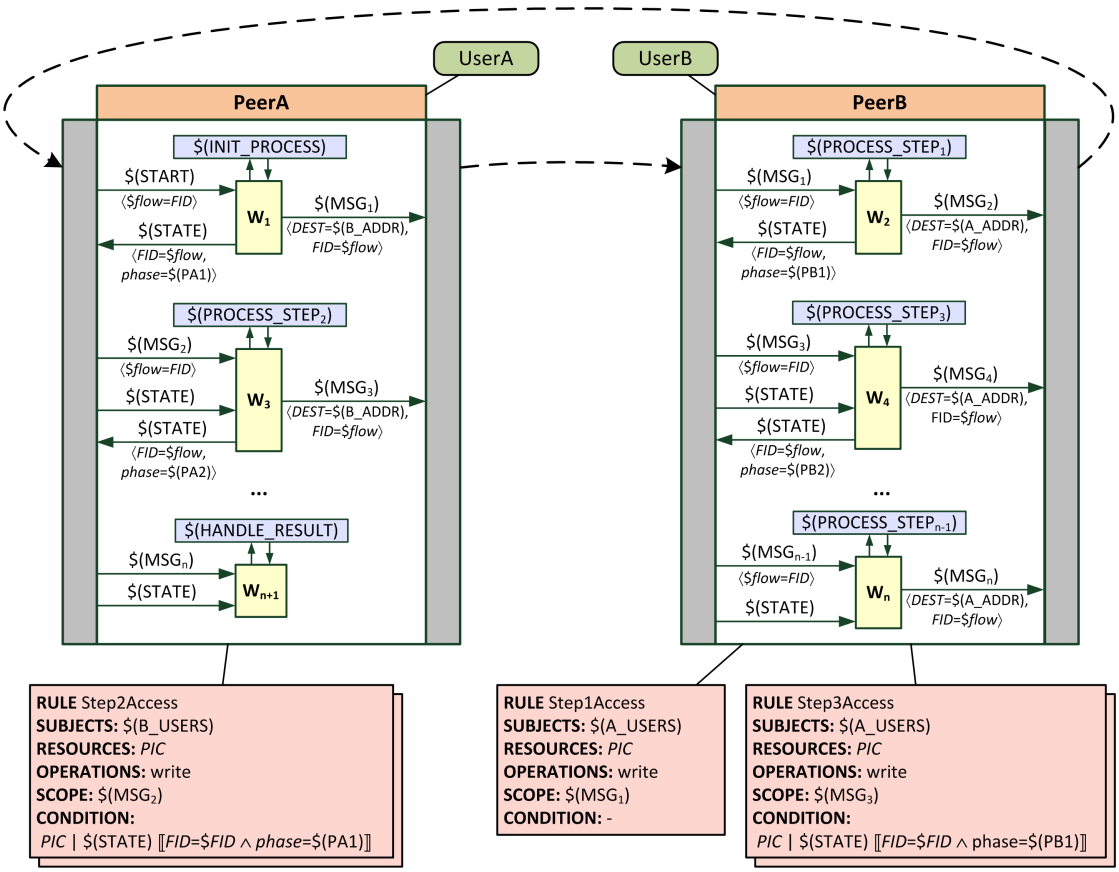

Figure 7.6: Solution model for Stateful Interaction pattern

mechanism ensures that a wiring instance always takes the right state entry, which belongs to the same flow as the received message. To achieve garbage collection, the last wiring on each site removes the local state from the PIC. In the shown scenario, *PeerA* processes the final message ($\texttt{MSG}_n$) and performs an appropriate result handling routine.

It is assumed that the start token does not require authorization, i.e., it is injected by the peer owner. For subsequent protocol steps, corresponding authorization rules are required. Starting a new interaction is always possible for authorized users. Therefore, rule `Step1Access` permits *UserA* (matched by `A_USERS`) to write entries of type $\texttt{MSG}_1$ into *PeerB*'s PIC. All following steps, however, depend on the current state. The remaining authorization rules on both sides follow the same structure: They permit the respective subject (matched by `A_USERS` or `B_USERS`, respectively) to write specific entries into their PIC, but only if the corresponding message type is expected according to the current protocol phase. For this, a condition is specified that checks whether a state entry with the right `phase` property exists for the given flow. The dynamic parameter `$FID` resolves to the flow ID of the entry to be authorized, i.e., the received message.

Variants:

- **Request-response correlation:** State information and phase-specific authorization rules may only be required on one side of the interaction, e.g., for a client that sends a sequence of requests to one or more servers. Intermediate results and relevant information about the current request can be stored within the client's state entry, so that the respective response handling wiring can correlate any response with its corresponding flow. In its simplest form, a flow only evaluates a single request. In contrast to the flow-based correlation variant of the Dynamic Response Handling pattern (Section 7.1.4), request-specific processing of responses is achieved by providing an additional input in the form of a state entry instead of requiring a customized service method.

- **Flexible order:** Depending on the selected interaction protocol, the order of messages may not be strictly defined. In some protocol phases, one of several different message types may be possible and some messages may appear in multiple phases. This can be achieved by extending the scope of a rule to cover multiple entry types and/or including a disjunction of possible phase values in the condition query selector. It may also be necessary to wait for multiple messages from different sources before the next phase can be reached. In this case, the corresponding wiring must block until all required entries are available, e.g., by using a suitable count specification or additional guards.

Consequences: The described pattern supports modeling of complex protocols with automatic correlation of protocol messages and peer states. The state entry acts as a local memory, which enables wirings to pass information to other wirings of the same peer that are invoked later within a flow. The utilization of the integrated flow correlation mechanism ensures that multiple protocol executions can happen concurrently, as each flow has a separate state entry that is automatically updated when a matching message of that flow is consumed.

The phase-dependent authorization policy ensures that messages only arrive when they are valid within their flow according to the defined protocol. Different subject templates may be defined for each step, e.g., based on the predefined roles of the protocol participants. The pattern enforces the local order of protocol steps within each peer. The global order of steps (e.g., in which order certain peers are invoked), however, cannot be easily controlled without a central coordinator. Due to the flow-based authorization rules, peers that are not participating in a flow cannot disturb the protocol by sending forged messages, unless they are already trusted by the peer owner (i.e., matched by the respective subject template) and able to guess the randomized flow ID. In some cases, security may be increased by writing state entries on behalf of the invoking user (e.g., using the subject "*RuntimeUser* **for** *UserA*" with a corresponding rule permitting this indirect access), whereas extended conditions additionally check whether the invokers of subsequent protocol steps match the stored principal for that flow (e.g., "*originator.userId* = $*invoker.userId*"). This way, hijacking of flows by other

authorized parties can be fully prevented. The presented solution model assumes direct access for message forwarding. Indirect access would also be possible, but then subject trees may become quite complex if multiple peers with different owners are involved.

Certain race conditions are possible, as multiple entries of the same type may arrive before their respective wiring, which updates the application phase and thus disables permissions for entries of this type within the flow, is executed. To prevent duplicate triggering of the wiring, a selector may be added to the guard link for the state entry, which checks whether the protocol phase is still valid for this step. If a flow should not wait indefinitely in case of lost messages, a timeout may be added to the corresponding state entry.

Applicability: This pattern provides another form of active access control as applied by workflow models. Process steps are only possible if corresponding prerequisite tasks have already been executed, thus supporting well-structured workflows and preventing inconsistent states. In contrast to the Context-Based Access pattern (Section 7.1.5), the relevant application state is automatically updated by the flow and not by an external source.

The pattern can be applied whenever a peer needs to retain its current state while waiting for a response from another peer. The state entry may include parameters of a request, an answer address, or other relevant values for future decisions. Possible use cases are consensus protocols like Paxos [Lam98] or atomic commit algorithms for distributed transactions (e.g., 2PC [Gra78]). For client-server scenarios, the pattern provides secure sessions that hold the current state of the respective interaction. Thus, stateless services as described in Section 7.1.1 may be extended to stateful ones.

The request-response correlation variant can be applied if a workflow involves a coordinating peer that queries other peers and triggers corresponding remote actions. In [CHKS14], a generic multi-master replication plugin for XVSM has been designed with the Peer Model. It uses a state entry to correlate a data retrieval request from a user with the result of a replica directory query and subsequently a list of matching entries from one or more replicas. The described authorization strategy may be used to protect the plugin from faulty responses.

7.1.7 Dynamic Workflow

Servers shall provide a controlled environment for the execution of workflows that involve injected behavior from different stakeholders.

Problem: Workflows may be composed in an ad-hoc way by connecting functionality of multiple tasks, where the output of one task acts as the input for another one. Instead of executing such workflows in a P2P fashion, a single server may host all involved tasks, thus providing a controlled execution environment. Clients dynamically inject and interconnect tasks on the server, which leads to the creation of workflows involving

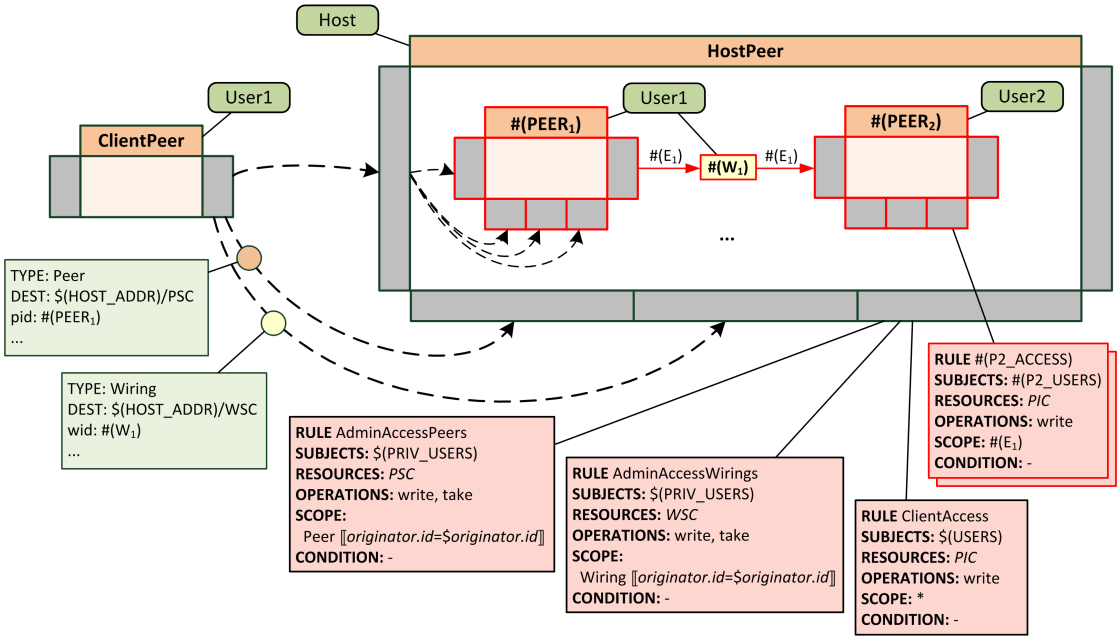

Figure 7.7: Solution model for Dynamic Workflow pattern

components from multiple stakeholders. After creation, these workflows can be triggered repeatedly by external users.

Regarding access control, three relevant roles exist. The hosting server shall be able to control any incoming or outgoing message, as well as the definition of workflows. It must be possible to modify or disable workflows that behave in a faulty or malicious way. Privileged clients shall be able to fully manage their own workflow parts (i.e., add, update, and delete tasks and connections), which includes the ability to define authorization policies for each task. Connections to tasks of other participants shall only be possible if the respective owner allows it. A regular client cannot define workflows, but it may trigger existing ones if authorized by both the server and the involved task owners.

Solution: A solution model with a simple, dynamically specified workflow is depicted in Figure 7.7. This example shows a host peer that acts as the central server for multiple clients, of which only one is shown. Workflow tasks are represented by sub-peers, while connections are built via wirings. A workflow can be started by sending a corresponding triggering entry to its initial task via the destination mechanism.

When starting up, the host peer is empty except for three initial authorization rules. Two of them allow administrative access for privileged clients (matched by `PRIV_USERS`) on the PSC and WSC, respectively. Peers and wirings may be inserted or removed following the owner-based access variant of the Shared Data Storage pattern (see Section 7.1.3), i.e., only the owner of a specification entry (or the respective originator in case of a compound subject) may remove it again. The rule on the WSC also covers the

insertion of a dynamic wiring that deletes or updates workflow elements, i.e., sub-peers and wirings. The last predefined rule addresses write access on the host peer's PIC, which is permitted for all authorized users (USERS). It is required for enabling external access to the sub-peers, as any entry that is targeted at a sub-peer has to pass this container. This applies to triggering entries as well as meta entries for administration.

Due to the Secure Peer Space architecture, the subject that has injected a sub-peer automatically has full permissions to set entries in its PSC, WSC, and SPC, thus defining its semantics. By default, no other subject (with the exception of the local runtime user) can access its internal behavior. In order to enable workflows involving different stakeholders, sub-peer owners must allow other subjects to access their PIC or POC. In the example, a client with subject *User1* injects a sub-peer ($\texttt{PEER}_1$) and a simple wiring ($\texttt{W}_1$) that connects its output with the input of an already existing sub-peer ($\texttt{PEER}_2$) owned by *User2*. As $\texttt{PEER}_1$ and $\texttt{W}_1$ share the same owner, the guard is implicitly authorized. For the action part, *User1* requires permission to write into the PIC of $\texttt{PEER}_2$, which has to be granted by its owner via a corresponding rule (P2_ACCESS) that is restricted to the expected input type ($\texttt{E}_1$). If $\texttt{W}_1$ were set by *User2* instead, it would require take permissions on the POC of $\texttt{PEER}_1$. The combined workflow, whose details are not relevant for the example, can be executed by sending a suitable triggering entry to $\texttt{PEER}_1$. In the shown configuration, only *User1* itself can start this workflow, but additional rules at $\texttt{PEER}_1$ may allow other subjects to trigger it.

Variants:

- **Indirect workflow definition:** Sub-peers, wirings, and rules are not directly manipulated by the involved workflow participants, but via a special management peer that indirectly accesses the PSC and WSC of the host peer on behalf of the originating user. This allows for a more controlled workflow execution, as the management peer controls which kinds of workflows are defined, while clients can still determine the behavior and permissions via properties within their request entry. In this case, the subject templates within the administrative and sub-peer access rules have to be updated to map the corresponding delegation chain, e.g., "*RuntimeUser* **for** *User1*". For sending triggering entries to a sub-peer, a PIC rule may be set that allows direct access by the originating user. Additionally, permissions to send requests to the management peer are required.

- **Indirect workflow execution:** Injected wirings may use indirect access mode, so that the full delegation chain of a workflow instance is preserved. In this case, the subject templates within the sub-peers' PIC rules have to match the compound subject. Each sub-peer may define own constraints based on the identities of the triggering user and the involved task owners. For allowing sub-peers to forward entries with compound subjects, additional POC rules are required similar to those used in the Stateless Service Invocation (Section 7.1.1) and Proxy (Section 7.1.2) patterns.

- **Information flow control:** Workflows may also communicate with external peers, e.g., to perform a query or to return a result. Wirings that emit entries with a set destination property require permissions to write to the host peer's POC. A simple form of information flow control is possible for the host by restricting which subjects may send which entry types to which other peers (based on their destination address). This can be achieved via a corresponding scope query, like "Response $[\![DEST = \text{'client1.example.com'}]\!]$".

Consequences: The central host server simplifies management, as workflow specifications are integrated at a single site. Participants do not have to message each other directly, which provides decoupling and reduces network overhead. However, a central server may also become a bottleneck for certain resource-intensive tasks that may be better solved in a distributed fashion. The presented approach allows for highly dynamic workflows, which can be adapted by external users or even via self-adaptation by sub-peers that modify the workflow by outputting dynamic wirings.

Due to the application of owner-based access (cf. Section 7.1.3), privileged users are able to manage their own tasks, but they cannot interfere with foreign tasks unless the respective sub-peer owner permits it. Wirings can combine tasks with different owners, but only if the subject that specifies the connection has the necessary permissions on source and target sub-peers. It has to be noted that it is still possible to install a wiring without having these permissions, as only WSC authorization is checked for this operation. However, its execution would never succeed.

Authorized clients trigger an installed workflow by sending a message to its starting task. In fact, external users may issue entries at any sub-peer, as the solution model does not distinguish between access from a remote client and access via a hosted sub-peer that was injected by the same user. This is usually not a problem, because for access to sub-peers that are not meant as starting tasks, any authorized subject has to be trusted to not disturb the workflow. Due to this feature, also a hybrid workflow execution model is possible, where different parts of a workflow are hosted on distributed servers.

In the presented solution model, the server retains full control of the hosted workflows. By dynamically changing rules in its SPC, the host can grant or revoke administrative access permissions and specify constraints on any incoming or outgoing messages. It is also possible to delete tasks and their connections by removing their specification entries from the PSC and WSC, respectively. On the other hand, sub-peer owners keep authority over the behavior of their tasks and the associated authorization policies. As already mentioned in Section 7.1.3, potential security risks emerge when allowing remote users to inject arbitrary wirings into a trusted peer. If these users are not fully trusted, the indirect workflow definition variant should be preferred for this reason.

Applicability: This pattern may be viewed as a generalization of the Shared Data Storage pattern (Section 7.1.3), as it enables a shared collaboration platform that not only hosts data, but also coordination and application logic. It can be compared with public cloud scenarios, where providers offer server infrastructure to host services of different

stakeholders while protecting them from unwarranted access. In [CJK15], a cloud-based request-response pattern is outlined, which combines elements of the Stateless Service Invocation (Section 7.1.1) and Dynamic Workflow patterns by hosting services within a cloud peer. Due to the multitenancy features of the Secure Peer Space, the Dynamic Workflow pattern enables the direct collaboration of cloud services from different users.

Mobile software agents form another related application field. Injected sub-peers can be compared to mobile agents in systems like LIME, as they contain specific behavior and collaborate with other entities on the same host. Dynamic workflows may also be useful for WSN scenarios, where resource-constrained sensor and actuator devices outsource the execution of their collaboration routines to a more powerful server. For the smart home management use case, different devices interact via a home server, while authorized users may specify workflows involving these devices.

7.2 Advanced Patterns

Using a limited number of basic patterns as building blocks, more advanced patterns can be constructed. This includes complex generic coordination tasks that require multiple interaction steps as well as highly specialized patterns that are only viable for specific domains. Solution models for such patterns can be derived from existing pattern solutions by using the following techniques:

- **Specialization:** A more specialized variant of a pattern is created by binding certain pattern parameters. They can be set to a specific value, like a domain-specific entry type or a concrete scope query selector, or partial binding may be applied in order to provide a more restrictive form of configuration. For instance, a generic subject template parameter may be replaced with a more specific subject template where only certain properties (e.g., the originator's role) are left configurable. Another way to simplify a pattern is to bind multiple pattern parameters together, e.g., by defining that two stakeholders shall always refer to the same subject, which may enable the simplification or omission of certain authorization rules by means of adaptation.

- **Adaptation:** Solution models may be slightly modified without changing the general concept of the pattern, e.g., by replacing a PMQ. In some cases, certain pattern elements may not be necessary and can be removed. In order to enrich the semantics of a pattern, the solution model may also be extended with additional components like peers, wirings, and individual links.

- **Composition:** Two or more existing patterns can be composed by merging their solution models. Peers from different patterns may communicate by exchanging entries or their coordination logic may be joined in a combined peer. Pattern parameters on each side (e.g., for entry types and subject templates) have to be attuned in order to provide a meaningful behavior.

The described pattern variants from Section 7.1 already outline the application of these mechanisms, but there are endless possibilities to create additional patterns by specializing, adapting, and composing basic patterns. Each of these patterns may in turn be used to construct even more advanced patterns. Thus, a wide range of patterns for different domains can be built whose solutions originate from only a few simple building blocks. Possible candidates for complex secure coordination patterns are:

- a secure lookup mechanism for locations of distributed Peer Spaces,
- master-worker schemes where tasks are distributed to multiple workers whose responses have to be verified,
- a publish-subscribe system with fine-grained permissions,
- reliable routing protocols for P2P networks,
- tamper-proof protocols for distributed consensus and distributed transaction commit,
- secure workflow execution with support for separation and binding of duty,
- a Peer Model equivalent of the Secure Service Space that allows dynamically specified services with different owners,
- and secured versions of coordination patterns for replication [CHKS14] and load balancing [CKBP14].

As outlined in [KCS15] and [Küh16], pattern configuration and simple compositions (without overlapping components) can be expressed by means of parameter bindings. For instance, "$(CONSTR) = (*val* > 1)" binds the CONSTR parameter of a given base pattern to a specific selector, while "*PatternA*.$(REQ) = *PatternB*.$(EVT) = $(MSG)" indicates that entry type parameters of two different sub-patterns refer to the same entry type (MSG) in the composed pattern. However, as currently no complete formal model for pattern derivations in the Peer Model exists that also covers adaptation and complex compositions, advanced patterns are linked to their base patterns in an informal way in the following example.

7.2.1 User-Specific Service Proxy

Based on a client request and stored profile information for the corresponding user, a proxy shall invoke a remote service on behalf of the user and transparently return its result.

Problem: Servers provide stateless services that can only be indirectly accessed by users via a proxy. Such a proxy stores relevant profile information for each user, which may include server preferences and other configuration options that affect service behavior. When a client sends a request, the proxy processes it based on the associated profile information and forwards a correspondingly adapted request to the dynamically determined server. The service response is then transparently returned to the client via the proxy. Concurrent invocation by multiple clients should be supported.

Requests have to be authorized by both the proxy and the server. Profiles are mandatory, i.e., no request shall be sent if it is not defined for the client user. Access control also has to check that only legitimate service responses are accepted.

Solution: Figure 7.8 shows the suggested solution model for this problem. Client peers send request entries (`REQ`) to a proxy RTP, which contains two sub-peers: a profile peer that stores the relevant user information within profile entries (`PROFILE`) and a process peer that performs the actual request processing. For that, the process peer retrieves the client-specific profile from the profile peer, uses this information to adapt the request (`REQ'`), sends it to a dynamically selected target server, and waits for the response. A locally stored context entry (`STATE`) is used for correlation with the received response entry (`RESP'`), which is then forwarded to the corresponding client as an entry of type `RESP`. To realize this solution in a secure way, several basic patterns are combined.

As a basic shell, the Proxy pattern (Section 7.1.2) is used. The process peer fulfills the role of the proxy peer from the original pattern. Instead of immediately forwarding the request, a two-step approach involving W_2 and W_3 is followed, which enables the incorporation of profile data into the decision process for the indirect server invocation on behalf of the client. The `ClientAccess` rule defines permissions for client users (`CLIENTS`) to use a specific request type. As several involved patterns require internal access by the local proxy user on behalf of the client, the `ForwardPermission` rule is generalized to an `InternalAccessPermission` rule that allows indirect write permissions on PIC and POC. This rule is instantiated for both the process peer and its RTP. At the server side, the RTP is not explicitly modeled. Therefore, trust- and permission-based rules of the Proxy pattern are merged into a single `ServicePermission` rule, which allows incoming request entries with a specific type for a set of authorized users (`AUTH_USERS`) as long as they have been sent by a valid proxy user (`PXY`) from a trusted runtime (`PXY_RT`).

The request-response variant for this adapted Proxy pattern is realized by applying two instances of the Stateless Service Invocation pattern (Section 7.1.1): for the client-proxy and the proxy-server interaction. Temporarily stored answer address properties (`replyTo`) are used to return the corresponding response entry to the correct destination, whereas the system variable `$$THIS_PEER` denotes the address of the own peer. As mentioned before, the server peer is not nested in this example, whereas the proxy RTP takes the role of the server RTP for the client-proxy interaction. Therefore, a general trust-based rule for clients (`ClientTrust`) is installed on the proxy RTP. Wiring W_1 uses impersonated access to forward each incoming request to the process peer

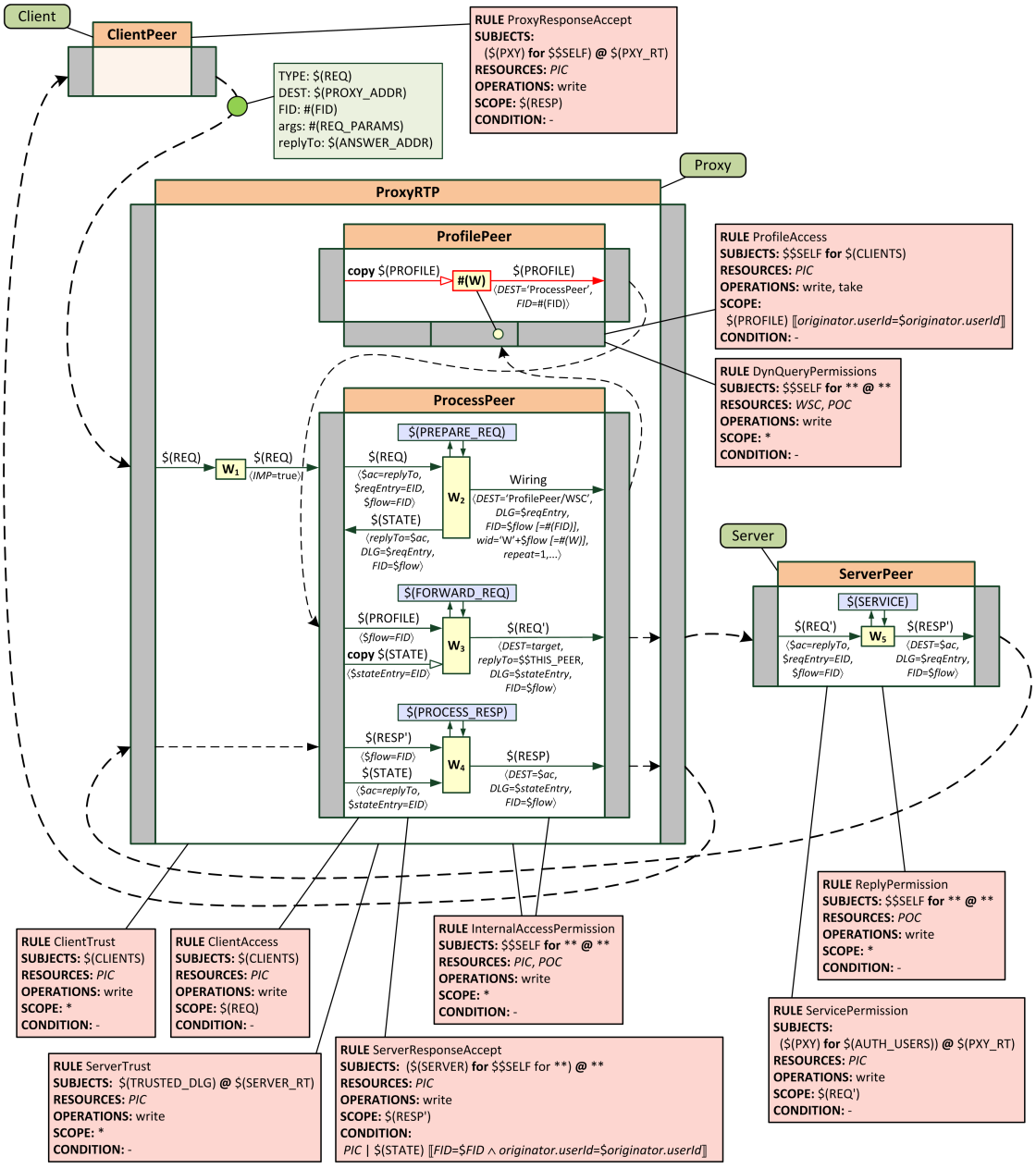

Figure 7.8: Solution model for User-Specific Service Proxy pattern

without changing its subject, so that it can be correctly matched by the aforementioned `ClientAccess` rule. Reply permissions for the proxy are already covered by previously described rules. For the server peer, another rule (`ReplyPermission`) has to be included for indirect internal access to the POC. Additionally, response acceptance rules must be established at proxy and client peers according to the Stateless Service Invocation pattern. The proxy only accepts responses from valid server users (`SERVER`) on a trusted runtime (`SERVER_RT`) that were sent on behalf of the own proxy user. Due to the nested architecture of the proxy peer, this permission is split into a trust-based (`ServerTrust`) and a permission-based (`ServerResponseAccept`) part. For the former rule, the `TRUSTED_DLG` parameter defines which kind of delegation trees are accepted from the server runtime (e.g., "** @ **" for full trust). Both rules together allow responses with the subject "(*Server* **for** (*Proxy* **for** *Client*) **@** *Proxy*) **@** *Server*". On the client side, knowledge about the involved server user is not required. Instead of using the response entry from the server as relevant entry for delegation at W_4 of the process peer, the corresponding state entry is used, which is stored after the initial request (by W_2) and therefore has the subject "*RuntimeUser* **for** *Client*". Thus, the request acceptance rule at the client matches response entries for own requests with the subject "(*Proxy* **for** *Client*) **@** *Proxy*".

The Shared Data Storage pattern (Section 7.1.3), in particular a combination of its indirect and owner-based access variants, is incorporated for accessing the profile peer within the proxy. However, instead of returning the result to the invoker, the profile data is used for the internal processing logic. After receiving a request, the process peer injects a dynamic wiring into the profile peer on behalf of the client, which reads the corresponding profile entry and returns it to the process peer. The scope query of the `ProfileAccess` rule ensures that only the profile of the originating client is visible to this wiring. The shown solution model only requires read permissions, but the rule also permits take and write operations. This ensures that additional wirings within the proxy RTP can write and update these profiles using indirect access mode, thus initializing the originator for each profile. To enable correlation of the retrieved profile with the stored state entry using the flow mechanism, the action of the dynamic wiring sets the flow ID to a dynamic pattern placeholder that corresponds to the `FID` of the respective wiring specification entry, which in turn is set to the `FID` of the original request by W_2. As the local proxy user can be fully trusted, the `DynQueryPermission` and `ReplyPermission` rules of the original pattern can be simplified into a single, less restrictive rule for internal access to the profile peer's WSC and POC.

The correlation of requests, corresponding profiles, and server responses is based on the request-response correlation variant of the Stateful Interaction pattern (Section 7.1.6). Using indirect access, wiring W_2 of the process peer stores relevant context information in the local state entry, which contains the client's answer address as well as relevant request parameters that may be set by the `PREPARE_REQ` service. As each involved entry shares the same flow ID, profile entries in W_3 and response entries in W_4 are automatically linked to the right state entry. In contrast to the original pattern, no application phases are used. Therefore, the state entry does not have to be updated

by W_3 and is only read. For security purposes, it is only relevant if a corresponding request exists for an incoming response entry. Therefore, a condition is added to the `ServerResponseAccept` rule, which ensures that a state entry with matching flow exists that also has the same originator as the response entry, i.e., the invoking client user.

Variants:

- **Profile updates:** After successful request handling, the process peer may dynamically update a client's profile, e.g., in order to incorporate statistical information. This can be achieved via a dynamic wiring emitted by W_4, which updates the user's profile via take and write operations on the profile peer's PIC. The overall authorization policy already allows such an internal access on behalf of the originating client user.
- **Multitenant environment:** There may be multiple process peers within a proxy RTP that address different kind of requests. Sub-peers could be associated to separate owners, which represent different applications or services. In this case, fine-grained permissions on the shared profile peer may specify which process peers are allowed to retrieve which types of profile information. Internal access rules for the proxy have to be adapted accordingly. Additionally, elements of the Dynamic Workflow pattern (Section 7.1.7) may be integrated to enable initialization and dynamic modifications of the proxy logic from different sources.

Consequences: The presented approach combines benefits of the involved basic patterns. Clients and servers are decoupled via an extended proxy architecture, where also the response is transparently forwarded to the invoker. Concurrent requests by multiple clients are isolated through the usage of a temporary state entry together with the flow correlation mechanism. Originator-based profile access authorization enables automatic association of a user request with the corresponding profile data because other entries remain invisible to the dynamic wiring. This user-specific information may cause request modifications (e.g., by adding properties depicting user preferences) or determine the server selection.

The coordination logic ensures that requests are only treated if a corresponding profile entry is available. To prevent that the flow blocks indefinitely when the guard of the respective dynamic wiring is not satisfied, timeouts may be used for state and dynamic wiring specification entries. Timeouts are also useful when the server does not respond. In such cases, the pattern may be enriched with suitable exception handling mechanisms in order to notify the client.

The authorization policies provide double protection for the server, as requests from malicious users can already be filtered out by a trusted proxy. They also prevent spoofed messages, as proxies and clients only accept responses when they were involved in the associated delegation chain. For the proxy, this is additionally verified by checking if a matching state entry with the same flow ID and the same originator exists. If clients do

not fully trust the proxy, they can employ similar strategies, as suggested by the Dynamic Response Handling (Section 7.1.4) or Stateful Interaction (Section 7.1.6) patterns. By locally storing the state entry with indirect access and using it as input for the final wiring, the round trip from the proxy to the server and back can be eliminated from the delegation chain of the final response, as the involvement of the server should be fully transparent for the client, also regarding access permissions.

Applicability: This pattern can be applied for realizing multi-tier architectures. For instance, proxy peers may represent Web applications, which often rely on stored profile information to request user-specific data from back-end or third-party sources. In the context of the examined firewall management use case, the SMC could issue requests to managed firewalls that depend on the account configuration of the invoking user.

The multitenant variant enables separate permissions for different proxy processes, both for accessing possibly privacy-relevant user information and for invoking remote servers. Even if the proxy is managed by a single administrator, the usage of different local system users for each process peer is still recommended, as it prevents unintended interference among application modules due to bugs.

Similar to the Secure Service Space, processes access local and remote containers on behalf of a client. However, the overall workflow is rather strictly defined, consisting of a local profile query combined with remote service invocation. In a more generalized form of the described pattern, more complex or even dynamically defined interactions could be supported.

7.3 From Patterns to Applications

The usage of patterns not only allows developers to select suitable solutions for specific coordination problems, in the long term it may also lead to a fully pattern-based development approach, where complex distributed applications are largely realized by composing and configuring existing patterns that are retrieved from extensive pattern catalogs. Such pattern collections may provide highly generic patterns, like in the given examples, or target specific application domains and use cases. For easier navigation, related patterns may reference each other, thus forming a so-called *pattern language* [AIS77].

Figure 7.9 visualizes this pattern-based development process. Solutions for new patterns can be built using Peer Model constructs like peers, wirings, and authorization rules, as well as solution models of existing patterns. In the design phase, developers can specify the basic architecture together with associated coordination and security constraints by combining several generic patterns with the help of some additional coordination logic acting as glue. The resulting design model can itself be interpreted as a pattern. This abstract application pattern can then be further refined by means of specialization, adaptation, and composition. Multiple refinement levels may be used that gradually replace pattern parameters with more specific values and include concrete application logic in the form of service code and external components. In the final

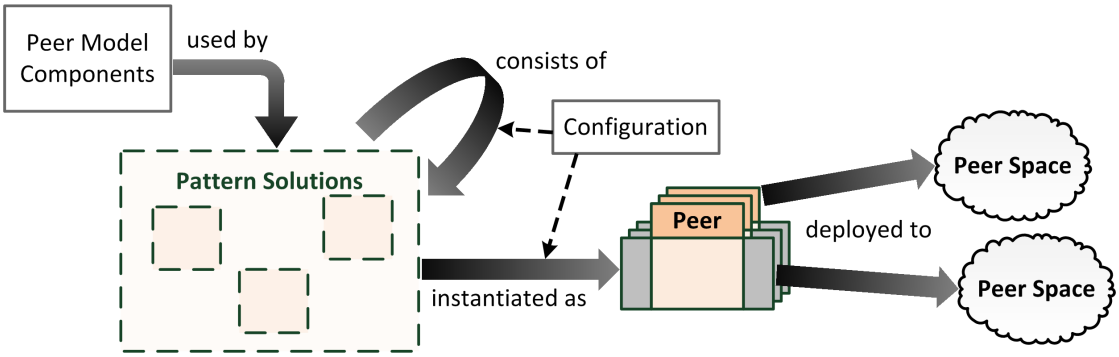

Figure 7.9: Overview of pattern-based development process (based on [KCS15])

application model, all pattern parameters are fully configured. Thus, it can be translated to a set of peer specifications that are deployed to one or more Peer Spaces.

In order to support such a development process, the existence of a usable toolchain is important, which integrates graphical modeling support for the Peer Model, querying and editing functionality for pattern languages, as well as flexible deployment options. Such an approach combines model-based development with the advantages of reusable pattern solutions, thus bridging the gap between pattern-assisted design and implementation. As applications are built on top of secure coordination patterns, developers not only model their behavior, but also their authorization constraints. This supports the creation of distributed applications that are secure by design.

CHAPTER 8

Applications

In order to demonstrate the feasibility of the suggested access control mechanisms, they have to be applied to realistic use cases. This chapter explains how complex distributed applications (as described in Section 3.1) can be implemented in a secure way by combining suitable coordination logic with corresponding authorization policies. As mentioned before, secure coordination patterns were gradually derived while designing the examined applications. For subsequent case studies, the identified patterns facilitate reuse of already tested solutions.

Two main examples are given to emphasize the characteristics of the two middleware technologies examined in this thesis. Section 8.1 describes a solution for the distributed firewall management scenario based on XVSM and the Secure Service Space architecture, whereas Section 8.2 shows how the Secure Peer Space can be used to realize the smart home management scenario. The examples address the overall software architecture and relevant authorization rules, while also highlighting connections to the examined secure coordination patterns. For both case studies, the feasibility of the given solution is analyzed, whereas possible adaptations and extensions are discussed. A full implementation of these extensive applications was out of scope of this thesis. Therefore, the evaluation puts a focus on the design stage, while important parts of the suggested models were validated via the implementation of corresponding test cases.

An overview of additional case studies is given in Section 8.3. The overall applicability of the embedded variant for WSN scenarios has already been discussed in Section 6.4, although specific LOPONODE use cases have not been examined extensively with regard to access control.

8.1 Security Management Center

For the distributed firewall management scenario from Section 3.1.1, an SMC has to be implemented that connects administration clients and managed firewall devices. Clients

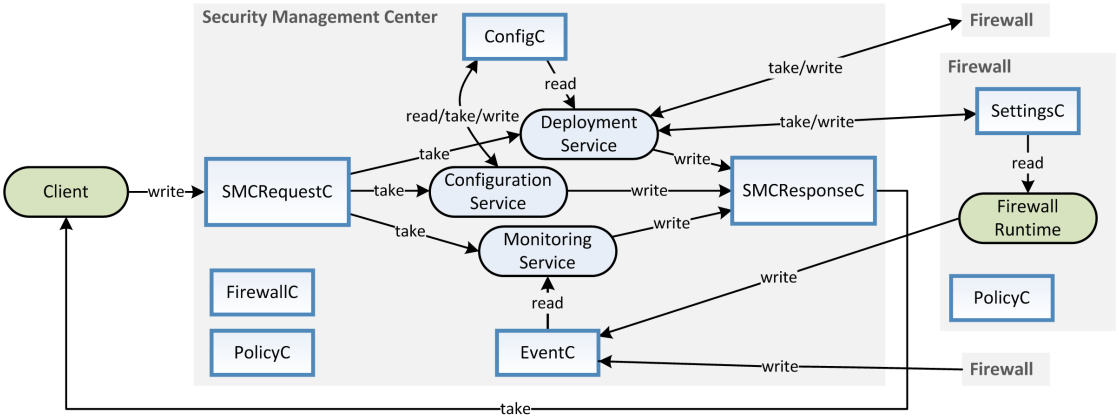

Figure 8.1: Simplified SMC architecture with XVSM (based on [CDJ+13])

shall be able to access relevant network events and configure firewalls at an abstract level. From the abstract configurations, the SMC can derive concrete settings and update the firewalls accordingly. As the SMC provides a wide range of functionality involving data with varying sensitivity, a fine-grained access control mechanism is required. Assuming a distributed administration setting, where multiple organizations share an SMC, also multitenancy has to be supported.

8.1.1 Solution

A simplified solution based on the Secure Service Space architecture is shown in Figure 8.1. Clients issue requests to the SMC's request container (*SMCRequestC*) and retrieve corresponding responses from its response container (*SMCResponseC*). The main functionality is realized by three services that access local and remote containers on behalf of the client user.

The configuration service enables clients to specify firewall configurations by means of high-level abstractions like templates and managed objects (e.g., representing a VPN tunnel), which can be assigned to a set of firewalls. All these data is organized in a tree-based data structure within the configuration container (*ConfigC*), which uses a special `path` coordinator that enables access to entries based on their specified path. For instance, a specific template may be stored at "templates/T42", whereas all templates can be retrieved at once using a wildcard, i.e., "templates/*". To assign a specific firewall *FW1* to a template, its identifier may be stored in an entry written to the container at the path "firewalls/FW1/template". Depending on the content of the request entry, the configuration service may add, delete, or update one or more entries in this data model using write and take operations within a local transaction. It is also possible to just query configuration data via read operations.

When invoked, the deployment service reads data from the configuration container to derive which firewall settings need to be changed. Then, it updates the remote settings containers (*SettingsC*) of the respective firewall devices, which adapt their behavior

accordingly. For each of its actions, the firewall runtime, whose internal behavior is not modeled, may look up certain parameters from its settings container and, if necessary, push relevant events or warnings to the SMC by writing them into an event container (*EventC*). The monitoring service allows clients to perform queries on this container in order to check the status of the managed devices.

For access control, the concepts of the Secure Service Space are applied, which enables service-specific as well as data-specific permissions and also protects response retrieval. Authorization rules are defined for the SMC and connected firewalls via their respective policy containers (*PolicyC*). The combination algorithm is set to `PERMIT-OVERRIDES`. Thus, the authorization policy is specified via a combination of only `PERMIT` rules. An additional container (*FirewallC*) holds registration entries for all managed firewalls, which provides relevant context information for the policy and can be set by privileged SMC administrators.

At the service level, permissions are defined per request type via rules on the SMC request container. For instance, the deployment of configurations may be restricted to users with a senior administrator role:

RULE DeploymentRule
SUBJECTS: [*role*: 'SeniorAdmin']
RESOURCES: *SMCRequestC*
ACTIONS: `write`
SCOPE: `type`(*DeployReq*)
CONDITION: -
EFFECT: `PERMIT`

More fine-grained permissions can be defined by targeting the indirect data access of the SMC services on behalf of the clients. The following rule allows indirect access to configuration data of selected firewalls on behalf of specific users:

RULE ConfigAccessRuleX
SUBJECTS: [*localAccess*: 'delegation', *DA.role*: 'Admin', *DA.affiliation*: 'X']
RESOURCES: *ConfigC*
ACTIONS: `write`, `take`
SCOPE: `path`('firewalls/FW24/**') `OR` `path`('firewalls/FW25/**')
CONDITION: -
EFFECT: `PERMIT`

The rule indicates that administrators from organization `X` can indirectly manage the firewalls *FW24* and *FW25* via SMC services. The used wildcard specifies that access to all sub-paths is allowed. Using more specific paths in the scope, access could be restricted to certain configuration options. Generic authorization rules can be defined by using a `label` selector within the scope that includes the affiliation attribute as a dynamic parameter. Then, any written entry must include a corresponding label and any query may only return entries that match the affiliation of the requestor. For instance, templates

may be tagged by the configuration service with the name of the organization that is responsible for them. This is demonstrated by the following rule, which additionally ensures that templates are stored at the correct location in the data model and have the right entry type:

RULE TemplateAccessRule
SUBJECTS: [*localAccess*: 'delegation', *DA.role*: 'Admin']
RESOURCES: *ConfigC*
ACTIONS: `write`, `take`
SCOPE: `path`('templates/*') `AND type`(*Template*)
`AND label`($*subject.DA.affiliation*)
CONDITION: -
EFFECT: `PERMIT`

In contrast to administration clients, firewalls directly access SMC containers, but they are only granted permissions to push their own events:

RULE EventPushRule
SUBJECTS: [*role*: 'Firewall']
RESOURCES: *EventC*
ACTIONS: `write`
SCOPE: `query`(*source* = $*subject.userId*)
CONDITION: *FirewallC* | `label`($*subject.userId*)
EFFECT: `PERMIT`

The scope prevents compromised devices from forging events of other firewalls, as each firewall must use its own ID as source property. The condition ensures that only registered firewalls, which have a correspondingly tagged entry in the firewall container, may push events to the SMC. In order to enable users to read these events via the monitoring service, additional rules for the indirect access to the event container need to be added. Permissions may depend, e.g., on the firewall ID and the event type.

Each managed firewall has to allow the update of its configuration via its settings container. In some cases, a firewall may restrict the set of trusted administrators independently from the SMC. The following indirect remote access rule depicts a case where the deployment service must act on behalf of a specific user (*User1*):

RULE SettingsUpdateRule
SUBJECTS: [*role*: 'SMC', *EA.DA.userId*: 'User1']
RESOURCES: *SettingsC*
ACTIONS: `write`, `take`
SCOPE: *
CONDITION: -
EFFECT: `PERMIT`

8.1.2 Analysis

Although the presented example does not provide a complete solution for distributed firewall management, the versatility of the XVSM access control model could be demonstrated. Due to the usage of the Secure Service Space architecture, the solution integrates concepts of the Stateless Service Invocation (Section 7.1.1), Proxy (Section 7.1.2), and Shared Data Storage (Section 7.1.3) patterns. Coarse-grained permissions are specified per service, while fine-grained authorization rules control indirect access to specific entries within the data model. Consistent firewall configurations are ensured as the devices may only be accessed indirectly via the SMC.

Access decisions may be based on entry contents, coordination data, and authenticated subject attributes. Although XVSM does not directly support the notion of entry owners, owner-based access is still possible by adding subject attributes as coordination data. Rules may also consider context information in the form of registration entries, which resembles the registration-based variant of the Context-Based Access pattern (Section 7.1.5).

Authorization policies may be dynamically changed by simply editing the policy containers of the SMC and the firewalls in order to reflect the current trust relationship among the involved stakeholders. Due to the transparency of the access control mechanism, clients and services do not necessarily have to be changed when permissions are modified. For instance, a request to show all accessible events may be realized via a read operation using the query "`any(ALL)`", which only returns entries for which the invoker is authorized, while other entries are automatically hidden.

Extensibility is supported, as additional features can be realized by adding new services that operate on existing containers, whereas their permissions are determined by the data access privileges of their invokers. The presented modules for configuration, deployment, and monitoring may be split into multiple services that cover more specific features, e.g., adding templates, querying firewall configurations, or generating concrete settings. Due to the modular and decoupled architecture, services and containers may even be distributed to different servers, which facilitates implicit load balancing and fault tolerance. Authorization policies can be easily adapted for such a case by permitting full access for server-to-server operations.

Based on the provided approach, more sophisticated solutions for an SMC may be modeled, using either the Secure Service Space or the Secure Peer Space. For instance, an advanced pattern for distributed transactions may be incorporated to guarantee the atomic deployment of configurations to multiple firewalls, or a distributed intrusion detection system may have access to special containers with anonymized event information from several SMCs.

8.2 Secure Workflows in Smart Home Environment

For the smart home scenario described in Section 3.1.2, a home server has to be realized that controls access to managed devices and orchestrates interactions among them.

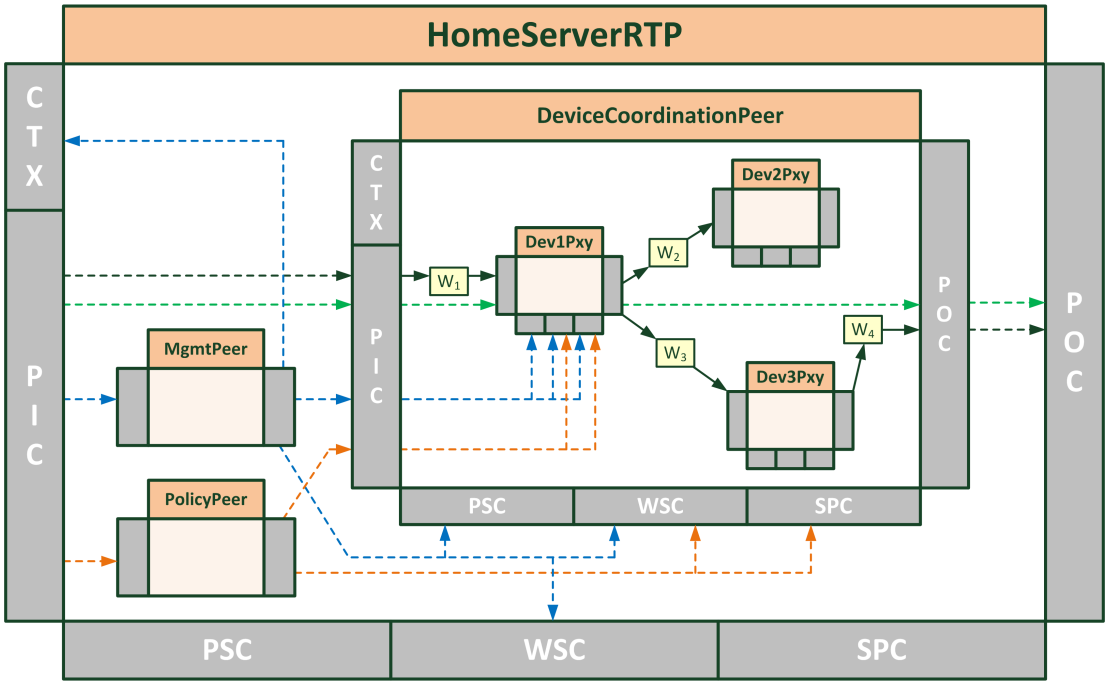

Figure 8.2: Architecture overview for smart home scenario with Peer Model

Administrators shall be able to add devices and model workflows that can be invoked by other users. Personal devices can be assigned to dedicated owners, who are able to control access to their functionality and their involvement in workflows. Thus, the home server has to enforce authorization for both human-to-machine and machine-to-machine communication. Corresponding rules may depend on the role of the subject, the accessing device type, and additional context information like the current time.

8.2.1 Solution

Figure 8.2 outlines a possible Peer Model architecture for a home server in this scenario. Its RTP hosts a management peer (*MgmtPeer*) for configuring devices and workflows as well as a peer that acts as a coordination room for devices (*DeviceCoordinationPeer*), where all interactions involving managed devices take place. An additional policy peer (*PolicyPeer*) enables configuration of authorization constraints. As the devices are in fact physically distributed and use heterogeneous communication protocols that are not aware of the Peer Space middleware, they are represented by proxy sub-peers within the device coordination peer. Each proxy provides an asynchronous interface for the communication with its associated device. It accepts a device-specific set of requests, invokes the corresponding functionality on the remote device, and handles possible results. It may also receive asynchronous events and cache relevant status information.

For each device type, own sender and receiver components need to be registered within the Peer Space runtime, which support communication with connected devices while

translating Peer Model entries into corresponding protocol messages and vice versa. The associated authentication manager must ensure that the set subject property correctly reflects the principal linked to the respective device. Using this approach, the proxies can communicate with their devices like with a remote Secure Peer Space.

The management peer offers services to add, update, and delete device proxies and workflows. Its wirings, which are not specified in detail here, emit corresponding entries to the meta containers of the device coordination peer and its sub-peers, while they use dynamic wirings to remove them again. In addition, they update corresponding registration entries in a separate context container (CTX). These interactions are depicted with blue arrows in the figure.

The configured device proxies can be invoked in two ways. Clients may directly address a specific device proxy (using an entry with type `Request`), which sends the corresponding response entry to the provided answer address (i.e., back to the client). This is indicated with green arrows in Figure 8.2. Clients may also trigger a previously configured workflow by sending a specific entry (with type `WfStartToken`) to the device coordination peer. In the example, black arrows are used to show this interaction. The workflow entry triggers wiring W_1, which invokes a specific function at *Dev1Pxy*. Corresponding result entries are written to its POC, from where they are picked up by subsequent wirings W_2 and W_3. Wiring W_4 may finally send a response using the destination mechanism. To enable the definition of different workflows, which all consist of a set of related wirings in the device coordination peer, start tokens include a corresponding workflow ID property (`wfid`). This property has to be passed along for all subsequently generated entries, whereas wiring guards use PMQs that only apply to entries related to their own workflow (e.g., "WfStartToken $[\![wfid = \text{'WF-1'}]\!]$" for W_1).

To prevent misuse, invoking clients as well as involved devices have to be authorized. Using the introduced access control mechanisms for the Peer Model, fine-grained permissions can be defined. For the sake of simplicity, only a part of the required authorization policy is described. Communication with the home server is allowed by managed devices as well as clients running on trusted control devices (e.g., a smartphone or PC). This can be expressed with the following rules at the RTP level:

RULE DeviceTrustRule
SUBJECTS: [*role* = 'Device']
RESOURCES: *PIC*
OPERATIONS: `write`
SCOPE: *
CONDITION: *CTX* | DevReg $[\![user = \$invoker.userId]\!]$

RULE ClientTrustRule
SUBJECTS: [*role* = 'Admin'], [*role* = 'User'], [*role* = 'Guest']
RESOURCES: *PIC*
OPERATIONS: `write`
SCOPE: *
CONDITION: *CTX* | DevReg $[\![user = \$invoker.devId \wedge ctrlDev = \texttt{true}]\!]$

The first rule ensures that only registered devices contact the home server, assuming that a corresponding registration entry exists in the context container. The usage of a separate CTX container prevents manipulation of context entries by external users, as they do not have permissions there. The second rule allows access by any client role, but only when using a trusted control device. In this case, the authentication manager has to verify the client credentials (e.g., a password) as well as the credentials of the used device (e.g., a certificate). The identity of the device is included in the subject via a corresponding property (`devId`). The condition ensures that this device is registered at the home server and marked as a control device (according to its `ctrlDev` flag).

Access to the management peer is restricted to users with the `Admin` role. Because it is owned by the local runtime user, this peer can use impersonated access mode to initialize device proxy peers and assign them to suitable subjects. As the proxy represents the actual device and not the administrator who triggered its creation, the peer owner is set to "*Server* **for** *DeviceX*". The former principal constitutes a local system user for the server logic with limited permissions, while the latter one is a new user representing the actual physical device, which is registered at the responsible identity provider. Administrators can specify whether a device is shared or belongs to a specific user. For shared devices, a default rule is initialized for access to the PIC of its proxy peer, which may look as follows:

RULE Dev1PxyUserAccessRule
SUBJECTS: [*role* = 'Admin'], [*role* = 'User']
RESOURCES: *PIC*
OPERATIONS: `write`
SCOPE: Request $[\![op = \text{'turnOn'} \vee op = \text{'turnOff'}]\!]$
CONDITION: -

This rule allows administrators and regular users to invoke the offered functionality of the device, which comprises two different services that are distinguished via the `op` property of the request entry. In contrast, personal devices by default only allow service invocation by the specified owner according to its user ID. This owner represents the responsible person for that device.

Additional permissions may be defined by the device owner (for personal devices) or any administrator (for shared devices). As clients should not directly manipulate the device proxy peer's SPC, these users are authorized to send permission update request entries to the policy peer, which updates the corresponding rules on behalf of the client using indirect access, whereas outdated rules are removed via dynamic wirings. Interactions with this peer are indicated with orange arrows in Figure 8.2. Each device proxy peer has to be initialized with a rule that permits such administrative access on the SPC (with subject templates of the form "*Server* **for** *DeviceOwner*" or "*Server* **for** *Administrator*", respectively). A possible request may demand guest access for selected features, while another one may include an ACL that explicitly specifies which users are allowed to invoke which device functions. Such request entries can then be easily transformed into corresponding rule entries.

In addition to these client-based authorization rules, also the access by the physical device to its proxy must be controlled. The corresponding device system user (based on its authenticated user ID) is therefore permitted to send selected entry types, which represent expected responses or events, to its device proxy peer.

When creating a workflow, the management peer injects the corresponding wirings with "*Server*" as owner, which leads to relatively simple subjects during workflow execution that still enable expressive constraints. Assuming that all involved proxy and workflow wirings use indirect access, the returned result of the depicted workflow in Figure 8.2 has a subject of the form "*Server* **for** *Device3* **for** *Server* **for** *Device1* **for** *Server* **for** *User*". Similar to managing permissions for device proxy peers, administrators are also able to control workflow invocations by indirectly accessing the SPC of the device coordination peer via the policy peer. The following example rule allows the invocation of a specific workflow by the parents of a family for a defined period:

RULE Wf1AccessRule
SUBJECTS: [*userId* = 'Mom'], [*userId* = 'Dad']
RESOURCES: *PIC*
OPERATIONS: `write`
SCOPE: WfStartToken $[\![wfid = \text{'WF-1'}]\!]$
CONDITION: *CTX* | Context $[\![\texttt{\$\$TIME}.hour \geq 8 \wedge \texttt{\$\$TIME}.hour < 21]\!]$

For the condition, the nested `hour` property of the system variable for the current time is checked. As the selector does not depend on any entry properties, a simple dummy entry with type `Context` in the context container is sufficient to activate the rule as long as the time matches. Similar rules can be defined when workflows should be started automatically by a device proxy based on an event received from its device (e.g., a burglar alarm triggering the activation of lights and a call to the police). Permissions may depend on the user ID of a device or on more generic security attributes like the device type.

The involvement of devices in workflows may be activated by allowing indirect PIC access ("*Server* **for** **") for the respective proxy peer, which is again set via a permission update request to the policy peer by an administrator (for shared devices) or a personal device owner. Using a restricted scope, only specific functionality could be made available for workflows. Permissions may also depend on the delegation chain within the subject. Thus, more specific subject templates may be defined. For instance, personal device owners could ensure that their devices can only be involved in workflows that were started by themselves ("*Server* **for** ** **for** *DeviceOwner*"). Similarly, also machine-to-machine communication may be regulated via corresponding rules that only allow invocations coming from specific device proxies ("*Server* **for** *DeviceX* **for** **").

8.2.2 Analysis

The presented example outlines how a sophisticated application involving different types of stakeholders can be realized with the Secure Peer Space. Its design is based on the

indirect variant of the Dynamic Workflow pattern (Section 7.1.7) with some adaptations to the handling of subjects. In the suggested solution, the physical device user, which is used as subject for messages coming from the device, has to be distinguished from the owner of a personal device, who may define authorization constraints. In order to decouple client and server implementations, administration is performed indirectly via authorized requests to management and policy peers. This ensures that the device coordination peer always provides a valid representation of the smart home environment.

The device proxy approach follows the overall concept of the Proxy pattern (Section 7.1.2), although the target server logic is not modeled with the Peer Model. The approach is also related to the Stateless Service Invocation pattern (Section 7.1.1), as it combines trust-based rules with service-level permissions in sub-peers. The registration-based variant of the Context-Based Access pattern (Section 7.1.5) is used to ensure that any accessing device has previously been added to the system by an administrator. Internally, proxy peer implementations may use additional patterns. For instance, the Dynamic Response Handling (Section 7.1.4) or Stateful Interaction (Section 7.1.6) patterns could be applied to block unexpected messages from users or devices, whereas locally stored status information may be retrieved using the indirect variant of the Shared Data Storage pattern (Section 7.1.3).

The presented architecture supports a wide range of authorization constraints that exceed the original requirements of an RBAC mechanism for device communication and workflows. It governs user access to specific device services and configured workflows, while also enabling fine-grained constraints for communication among devices. However, it has to be noted that the home server can only manage interactions that are configured within the device coordination peer, as direct P2P communication between devices is not controlled by the system.

A major challenge while modeling such complex applications is the assignment of suitable subjects for each interaction. Including additional subject information allows for even more fine-grained constraints, but also increases the complexity of the modeled policies. For instance, the identity of the responsible administrator could be added to the subject for the definition of workflows and device proxy peers, which would facilitate the creation of separate security domains within a single home server. Additionally, the usage of separate system users for different parts of the server logic (policy peer, device proxies, workflow wirings, etc.) would be beneficial, so that each component only possesses necessary privileges. This serves as a protection against bugs and may prevent privilege escalation in case of compromised services.

8.3 Further Case Studies

Besides the firewall management and smart home scenarios, the access control mechanisms for XVSM and the Peer Model have also been evaluated with regard to several additional use cases, which were part of smaller case studies and student projects. Relevant examples include:

- Access control for a simple conference management system has been modeled with the Secure Space in [CDJK12]. Authors may only access data related to their own submissions, while reviewers can only read assigned papers. The solution supports context-based access, as authorization depends on the current phase of a conference and the existence of a program committee registration entry for any reviewer, both of which can be edited by a program chair.

- Schmid [Sch13] describes an XVSM-based solution for automatic service migration among different cloud platforms, which involves P2P coordination between multiple clouds and their customers. The suggested security mechanism is based on the Secure Service Space architecture. Context-aware authorization rules ensure that customers can only access their own cloud services and that certain operations are only valid during an active migration.

- Kanev [Kan15] uses the Secure Service Space to implement a P2P video streaming framework. Authorization is enforced on the video meta data, which is stored in a central component. Videos can only be deleted by their owners, who can also block specific viewers. Additionally, videos may be billable, so that users can only access them if they have enough points in their stored wallet entry.

- The micro-room framework by Binder [Bin13, KCBŠ19] facilitates the easy definition of P2P applications by specifying workflows involving configurable coordination modules (i.e., micro-rooms). It is built on top of MozartSpaces and also applies the Secure Service Space architecture for access control. Authorization rules are abstracted via plugin settings, which support fine-grained permissions within each micro-room, where access to each offered service can be granted based on users, roles, room membership, room ownership, and ownership of affected data items. The feasibility of this approach is demonstrated by realizing a privacy-aware P2P social network for e-learning.

- Bitter [Bit15] evaluates his .NET implementation of the Secure Peer Space in a case study that addresses the management of student assignments. Permissions are context-aware, as they require prior registration by the users and distinguish different phases. Students can submit solutions to a lecture server if the entries are marked with their own registration number. The lecture server forwards the solutions to assigned tutors for grading, which respond with a grading proposal. Finally, the supervisor only accepts grading proposals from the lecture server when it acts on behalf of a tutor.

- Using the Java implementation of the Secure Peer Space, Lettmayer [Let18] has implemented a P2P volunteer computing framework, which enables project owners to split complex computational tasks into sub-tasks that are sent to distributed workers. To share their computing resources, users search for relevant projects and register to them. Only accessible projects are visible, as project owners can include or exclude users based on specific attributes, like country or number of collected

points. The authorization policy ensures that only the assigned worker may send a solution for a given sub-task. Moreover, also workers can dynamically deactivate acceptance of sub-tasks for certain projects, e.g., to prevent overloading.

CHAPTER 9

Evaluation

The main objectives of this thesis were the design of flexible access control models for space-based middleware, the specification of secure middleware architectures that are able to enforce them, and the promotion of reusability for secure coordination solutions by means of patterns. Based on the requirements defined in Chapter 4, this chapter describes how these goals have been accomplished with the introduced concepts and systems, while it also discusses remaining open issues. Section 9.1 evaluates expressiveness and usability, while providing a feature comparison with related work. Section 9.2 analyzes the security of the presented approach, whereas Section 9.3 deals with its practical feasibility.

9.1 Expressiveness and Usability

Most of the compiled requirements target expressiveness and usability of the access control model. They indicate the need for sophisticated yet comprehensible authorization policies that provide a suitable abstraction for the underlying distributed coordination model and can be managed in a flexible way. In the following, the fulfillment of these requirements is examined for the introduced secure versions of XVSM and the Peer Space.

Fine-grained access control (REQ-1) is ensured as authorization is evaluated and enforced at the level of individual entries. Content-based access control (REQ-2) is realized via the scope concept, which enables the selection of accessible entries based on dynamically evaluated queries that target certain entry properties. Context awareness (REQ-3) is facilitated via conditions and dynamic parameters. Expressive condition queries enable permissions that depend on the existence and content of special entries depicting an application state or environmental context information, while dynamic parameters integrate the request context into authorization rules. Following the ABAC paradigm, a proper subject abstraction (REQ-4) is provided based on matching security attributes. XVSM offers a simple template matching approach, whereas the Secure Peer Space supports arbitrary predicates to describe principals in a symbolic way.

The completeness property (REQ-5) is fulfilled as the relevant middleware functionality can be reduced to container access operations, for which permissions can be specified in a unified way. Service invocations are covered by treating requests and responses as regular entries. Consequently, each function offered by the middleware itself can also be mapped to a specific container access. For XVSM, the Secure Space architecture supports invocation-based rules for any CAPI request. In the Secure Peer Space, the coordination logic can be modified via the manipulation of entries in meta containers, while it may be possible to invoke additional functionality by sending request entries to special system peers.

As access control is managed independently for each middleware runtime via local policy containers, decentralized authorization (REQ-6) is supported. Authorization not only covers direct access to entries, but also indirect data access by services on behalf of their invokers (REQ-7). For XVSM, this is enabled via the Secure Service Space architecture, while the Secure Peer Space provides an integrated solution that also allows for chained delegation. Tree-based subject templates with wildcard support enable the definition of trust-related rules based on attributes of involved principals at defined positions within a flow. This constitutes a distinctive feature that cannot be found in other middleware systems, which mostly disregard compound subjects completely. Practical security frameworks may enable indirect access by letting services run in the context of the invoker, but usually only impersonation is supported (i.e., the identity of the invoking user is used for authorization) instead of complex delegation chains that could be useful for workflows involving multiple stakeholders.

Regarding federation (REQ-8), both access control models specify conventions for a generic representation of identities that is not bound to a specific authentication mechanism. Subjects from different organizations are distinguished via affiliation or domain properties, respectively. The Secure Peer Space additionally supports an authentication context that includes information about the used identity provider. Trust in the validity of subject information may be specified via trust-based rules that rely on the authenticated domain, the authentication context, and the identities of any runtime that has forwarded the respective entry. While authentication of federated identities was not explicitly addressed in this thesis, it can be integrated by implementing corresponding authentication modules (e.g., based on SAML).

The support for multitenancy (REQ-9) varies among the introduced access control models. In the Secure Peer Space, coordination logic from many stakeholders can be co-located on the same space. Each peer has a dedicated owner that is responsible for managing permissions. Due to the hierarchical policy structure, nested security domains are supported. Collaboration between different security domains is possible by installing suitable wirings, as long as their container accesses are authorized by all involved stakeholders. An XVSM space only supports a single security domain by default. Different processes are still able to collaborate, but the security of the interaction is more dependent on the responsible space administrator, who has to set up suitable permissions for each participant. Additional security domains could be added manually via respective administrative rules that delegate the permission to define rules for specific containers,

but such an approach is not recommended due to the increased complexity of policy management.

Dynamic policy changes (REQ-10) are possible by modifying rule entries in policy containers. Combined with the decentralized authorization architecture, this ensures applicability for dynamically evolving P2P applications with multiple stakeholders. Administrative rules (REQ-11), which enable the (partial) delegation of rights to remote administrators, are implicitly supported via rules on the own policy container.

Coordination logic and access control are clearly connected, but they can still be modified independently from each other. Thus, a reasonable level of decoupling (REQ-12) is achieved, which fits the space-based interaction style and ensures separation of concerns. Transparent access is facilitated by authorization-aware queries that filter out inaccessible entries. This enables *authorization-aware coordination*, where coordination logic requests a specific set of entries, but the authorization policy determines which of these entries are actually visible for the requestor and may be returned. Thus, permission changes may affect the outcome of an operation, but do not require updates to the coordination logic as long as no additional information is required by the authorization rules (e.g., in the form of context entries). Similarly, changed coordination code does not necessarily require additional authorization rules unless the container structure is modified or new entry types are introduced.

The presented access control models provide concise policy languages with relatively simple syntax and semantics (REQ-13). For XVSM, a flat policy structure is used with a configurable combination algorithm, while the Secure Peer Space applies hierarchical policies with fixed combination semantics. For most use cases, such a simplified approach seems to be sufficient. Administration is more complex compared to using basic ACLs, but considerably simpler than applying general-purpose policy languages like XACML. As access control has been customized for XVSM and the Peer Space, respectively, a suitable abstraction for modeling secure collaboration could be found, which provides a reasonable compromise between expressiveness and comprehensibility. The bootstrapping requirement (REQ-14) is fulfilled due to the usage of already established middleware concepts, like XVSM queries and PMQ selectors within authorization rules, as well as meta containers for managing rule entries. This approach helps developers that are already familiar with the middleware to adapt quickly to the secured versions. It also facilitates *coordination-aware authorization*, which means that permissions can be tightly aligned with the expected interactions, as both rely on common coordination principles.

Based on the previous analysis, the secured versions of XVSM and the Peer Space can be integrated into the classification of secure middleware technologies that was performed in Section 2.2.4. Table 9.1 compares them with a selection of the most relevant related systems according to their assessments from Table 2.1. Both middleware technologies obviously enable flexible coordination by means of their comprehensive query features and support for reactive programming via aspects and wirings, respectively. The previously targeted requirements cover the remaining evaluation criteria regarding expressiveness and usability of the access control model. The evaluation shows that the secure versions of XVSM and the Peer Space compare favorably with related systems.

	Flexible Coordination	Fine-Grained Permissions	Subject Abstraction	Context Awareness	Decentralization	Administrative Policies	Usability
TuCSoN	+	+	∼	+	+	+	∼
LGL/LGI	∼	+	+	+	∼	−	∼
EgoSpaces	+	+	+	+	+	−	+
GigaSpaces XAP	+	∼	∼	∼	+	∼	+
Triple Space	+	−	+	−	∼	∼	∼
SALSA	+	∼	∼	∼	+	−	+
SECTET	+	∼	+	+	∼	+	∼
CHOReVOLUTION	+	∼	+	+	∼	∼	−
Hermes	∼	+	+	∼	∼	+	∼
Secure XVSM	+	+	+	+	+	+	+
Secure Peer Space	+	+	+	+	+	+	+

Table 9.1: Secure middleware comparison with secured XVSM and Peer Space versions

Also compared to conventional access control frameworks for distributed systems (e.g., Spring Security), several advantages can be identified. Due to the limited number of features, which are custom-tailored for the underlying coordination model, relatively simple access control models could be devised, in contrast to cluttered general-purpose frameworks that support many configuration options. In classical security architectures, authorization is mostly used to protect files and services, but volatile coordination entries cannot be targeted if the corresponding middleware runtime is treated as a black box. For instance, without integrating authorization directly into the query mechanism, fine-grained content-based permissions would not be possible. Authorization rules on incoming operation requests could restrict possible query parameters, but queries do not necessarily target properties that are relevant for access decisions (e.g., when random access is requested). On the other hand, filtering authorized result entries after query execution would already be too late, as the operation may have had side effects, like the removal of entries from a container. Therefore, an integrated approach has been followed that enables higher expressiveness. This ensures that a wide range of security constraints can be specified in the authorization policy instead of requiring implementation of complex conditions directly in the application code. Additionally, the bootstrapped policy management enables flexible permission changes without modifying configuration

files or code annotations. This also supports highly adaptive solutions, where intelligent components are able to dynamically change authorization policies.

In summary, it can be argued that the presented access control models provide suitable mechanisms to specify permissions for distributed applications consisting of autonomous components. Expressive yet comprehensible policies can be specified in a flexible way via a bootstrapped approach that integrates security concerns with the data-driven coordination mechanisms in a natural manner. However, in order to further improve expressiveness and usability, some open issues still have to be addressed. The query languages are not yet fully optimized. For instance, nested policy sets could be introduced for XVSM to enable a more structured policy management, while the definition of subject templates for delegated access in the Secure Peer Space may be simplified by providing shortcuts for common constraints (e.g., for single-tenant scenarios where each peer owner is also its own runtime user). Additionally, special system peers and graphical administration tools may enable the definition of policies at an even higher abstraction level, where users select a predefined strategy (based on a secure coordination pattern) and just have to provide the required parameter values, whereas the resulting rules are generated automatically and put into the right containers. Additional research is also required for the integration of existing technologies for federated identity management and authentication. Finally, a formal specification of the secure middleware versions is necessary to fully clarify access control semantics in any case.

9.2 Security Analysis

The introduced access control models follow state-of-the-art security recommendations that were observed in literature and related middleware solutions. Expressive authorization policies ensure that users are only given necessary permissions. Furthermore, active access control is possible, where permissions are only valid when a corresponding coordination step is actually expected. As demonstrated by the suggested secure coordination patterns, this expressiveness enables administrators to satisfy the principle of least privilege.

The ABAC approach enables flexible policy management for open distributed environments. By means of role attributes, also RBAC is supported, which is widely used in enterprise settings. As permissions rely on assigned security attributes, they automatically expire when an employee leaves an organization. In the Secure Peer Space, also elements of DAC are integrated, as peer owners can specify their own authorization policies, even if they are hosted at a foreign runtime. Isolation between such independent security domains is ensured, as no interaction or policy change is possible without explicit permission from the owner (except by the local runtime user).

The delegation support allows for permissions that depend not only on the direct invoker, but also on other involved subjects. However, in contrast to related delegation schemes like those of PERMIS or DSSA, the delegation process is not actually verified with cryptographic measures, although some authentication variants may include this feature, as demonstrated by the secure routing mechanism for LOPONODEs described

in Section 6.4. Such a mechanism only works when messages are directly forwarded via a chain of one or more intermediate nodes. However, in the Peer Model and the Secure Service Space architecture, workflows typically consist of different kind of entries that are processed, merged, or modified in each step. It might be possible to piggyback the signed initiating entry onto the subject information of an entry, but then any receiving entity would have to determine if the received entry is a valid consequence of a delegation triggered by the attached entry, which would basically require a duplication of the logic of all intermediate nodes.

Instead, a simple delegation mechanism based on explicit trust assumptions has been adopted. Users implicitly authorize a peer (or service) to act on their behalf by sending entries to it. This follows the decoupling principle of space-based computing, as the originator does not require any knowledge about the inner workings of the invoked peer, which may perform a requested operation on its own or call one or more additional peers. At the receiving side of a delegated entry, only the credentials of the actual sender are verified. Using expressive subject templates, administrators can specify which type of subjects are trusted and which permissions should be assigned to them. For XVSM, this means checking if the delegated attributes are consistent with the authenticated identity, while for the Secure Peer Space, the credibility of the provided subject tree has to be evaluated according to the trustworthiness of the authenticated runtime user and the other principals within the claimed delegation and authentication chains. Thus, the policy language combines trust-related and privilege-related rules into a single rule type.

The presented access control models supports multiple layers of protection (REQ-15), thus providing defense in depth. In the Secure Service Space, users first have to be permitted to perform XVSM operations on a space, then to invoke certain services, and finally to indirectly access specific data via these services. For the Secure Peer Space, nested protection applies both locally and globally. Within a single space runtime, the hierarchical policy structure enforces the specification of permissions at each peer level. Access to sub-peers has to be allowed also by the owners of parent and ancestor peers, including the runtime user. Within a distributed flow, each involved runtime may act as a gatekeeper that autonomously assesses the trustworthiness of an entry. A flow can only succeed when its individual steps are permitted by all involved participants. This defensive approach significantly reduces the risk that a single misconfiguration compromises a system. High-level peers, especially those involved in the early stages of a flow, may grant rather general permissions, whereas policies should become more and more restrictive towards the actual location of sensitive services and data.

These security concepts as well as best-practice solutions in the form of secure coordination patterns reduce the risk of configuration errors, thus increasing the dependability of implemented applications. However, secure collaboration is only possible when the involved middleware runtimes also effectively enforce the specified access control models (REQ-16). The respective middleware architectures have therefore been designed in a way that can be mapped to the well-established XACML authorization framework with PEP, PDP, PAP, and PIP components. The system intercepts all relevant operations and only executes them if they are permitted. Access decisions solely depend on local

authorization policies and context information, which are protected with own mechanisms. Thus, the security components can be implemented as part of a TCB that is protected from a wide range of attacks. In the following, possible attack vectors and corresponding countermeasures are outlined:

- **Malicious code injection:** Injected code that is executed by the middleware runtime may bypass security mechanisms or cause other problems like performance losses or crashes. For the Secure Peer Space, therefore only trusted services shall be invokable by wirings. These could be set by privileged administrators via a special system peer. In XVSM, aspects can potentially access any local container without authorization checks. Therefore, remote users shall only be able to install a selection of preconfigured aspects. If services or aspects with arbitrary application logic are required, strict sandboxing mechanisms have to be added to the respective middleware architectures to ensure isolation and provide hooks for authorization regarding local resources like databases.
- **Unsafe object references:** Conceptually, entries within a space are strictly isolated from each other. However, some runtime implementations may allow shared references for copied entries or entry properties. Thus, changes to one object also affect the other object while bypassing access control. When an entry is read from a local container, the accessing service must not be able to modify the original entry in the source container, but only its copy. Similarly, when writing an entry to a local container, the invoker must not be able to manipulate the written entry by subsequently changing property objects. A feasible solution is to create deep copies of potentially affected entries, even though this may cause a significant performance overhead for local operations.
- **Spoofing:** Malicious runtimes may provide fake subject information and thus access otherwise denied containers. A runtime could use valid credentials for its own identity, but fabricate information regarding delegation and/or authentication chains. Due to the lack of cryptographic measures for delegated attributes, this cannot be fully prevented, but the problem can be largely mitigated via trust-based rules that only allow specific types of delegations depending on the established trust level. Nevertheless, privilege escalation should not be possible, as the authenticated user always remains part of the subject, assuming that the authorization policy does not ignore the corresponding security attributes in the relevant subject templates. Spoofing must also be prevented for responses, which are not authorized in XVSM. Therefore, the receiver component has to ensure that the referenced request ID within a response corresponds to the randomly assigned ID of an active request and its source equals the request's target space URI. For the Secure Peer Space, authorization rules as suggested by patterns like Dynamic Response Handling (Section 7.1.4) can prevent unsolicited responses.
- **Byzantine faults:** Clients or runtimes may suddenly behave in an erroneous or even malicious way, e.g., by emitting arbitrary events or returning wrong results.

This is especially problematic if the associated subject is authorized to perform such operations, as access control cannot decide if a given interaction is semantically correct according to the desired application logic. This issue can be solved by using redundant nodes in combination with a Byzantine-fault-tolerant consensus mechanism, which ensures that correct processes eventually agree on their output as long as there are not too many faulty processes. Such a distributed consensus algorithm may be specified as a secure coordination pattern that can be applied to different application scenarios. Users with detected Byzantine behavior could be automatically blocked by dynamically changing the corresponding authorization rules.

- **Covert channels:** Certain side effects of the access control mechanism may allow users to learn about properties of denied entries. The applied transparent access semantics prevents users from getting information about hidden entries based on authorization error messages. However, if rules with conditions are used, the status of context entries can be assumed based on the success or failure of specific access operations. Rules have to be designed in a way that sensitive information is never revealed via conditions. Therefore, only fully trusted subjects should be given the permission to specify authorization rules. This is also the reason why conditions are restricted to the own peer in the Secure Peer Space, as otherwise peer owners may be able to indirectly query entries from other security domains.

- **Race conditions:** Inconsistent states may occur when authorization checks and policy changes happen concurrently. Therefore, policy changes in XVSM should always be performed atomically by including all related operations on the policy container in the same transaction. Similarly, also the atomic update of an SPC in the Secure Peer Space can be ensured via a single wiring. Changing several authorization policies within the hierarchical peer structure may require multiple steps. As only "permit" rules are possible and policies remain consistent at each hierarchy level, this should not enable unauthorized access, though. Due to the transactional access mechanism, individual authorization checks are always based on a consistent policy state, so either the new or the old policy applies. However, authorization rules may be changed during operations that involve multiple container accesses, like the execution of a wiring with two or more links. To prevent such concurrent policy updates, short-term locks on policy containers can be used. Longer-running interactions like flows could still be interrupted when required permissions are revoked, which may lead to inaccessible entries that should eventually be removed (e.g., automatically via TTL), but this is not a security problem per se.

- **Revoked privileges:** The revocation of privileges has to be handled properly, as obsolete permissions and coordination logic may persist otherwise. When administrative privileges are revoked from a user, all rules owned by this subject should be reevaluated and potentially removed in order to prevent the existence of backdoors. Similarly, also installed sub-peers and wirings should be removed if

the respective permissions of their owners are revoked. Due to the lack of an entry ownership concept, an alternative approach is necessary for XVSM, e.g., based on an accountability mechanism that tracks relevant configuration changes by any user.

- **Obsolete security attributes:** Security attributes associated to users may change over time, either due to general changes at the identity provider or factors related to individual accounts. Authorization rules must not target attributes whose names or semantics are volatile, as any change by the identity provider could lead to inconsistent behavior, e.g., when an expected attribute is suddenly missing. When security attributes of a specific user account are modified (e.g., after a department change), the configured authentication mechanism shall return the updated identity for subsequent space accesses. However, a special handling is required for injected wirings, which assume a snapshot of the owner identity at the time of their installation. Thus, they may still be able to access entries even though their owner has already lost the respective permissions due to changed security attributes. A simple way to address this problem is to allow only wirings with a set TTL, so that they have to be refreshed in regular intervals using up-to-date owner identities.

- **Man-in-the-middle attacks:** Different type of vulnerabilities related to the interception and/or illicit forwarding of messages have to be addressed. Eavesdropping of sensitive information or hijacking of interactions (i.e., where an attacker intercepts communication to a specific space runtime and returns manipulated responses) must be prevented by using secure communication channels that provide end-to-end security. Additionally, malicious runtimes must not be able to impersonate other principals by replaying received messages or forwarding provided credentials. Authorization policies may restrict repeated replays of the same entry (e.g., via one-off rules tied to specific flows), but full protection against impersonation requires additional measures within the authentication module that confirm the true source of an entry.

- **Compromised identity providers:** Malicious identity providers would be able to assign arbitrary security attributes to any user, thus circumventing access control, which relies on the correct authentication of remote principals. Therefore, these servers have to be particularly protected from any attack. In general, a runtime should only use identity providers that are managed by a fully trusted organization (e.g., the own one).

- **Software vulnerabilities:** Hackers may be able to bypass security mechanisms by exploiting vulnerabilities in the middleware runtime implementation, the server application running the space (which has full access to it), or the hosting environment (e.g., an OS or a virtual machine). This can only be prevented via a rigorous development process with extensive tests as well as continuous security updates.

All in all, the presented concepts describe effective measures to ensure confidentiality and integrity of data. However, the actual level of security largely depends on the quality of available runtime implementations and the correct usage by involved administrators. The current middleware prototypes act mainly as proof-of-concept for the introduced access control models. The next step would be the implementation of robust middleware runtimes with a TCB that cannot be bypassed. The integration of standardized technologies like SASL, SAML, or TLS may improve reliability and security. Furthermore, the validity of the formalized access control models in different situations shall be thoroughly examined via model checking. In order to avoid configuration errors, the usability of the policy management process can still be improved, as suggested in the previous section.

9.3 Practical Feasibility

The proposed access control models and their respective coordination middleware can be utilized in different ways. For one, they enable the design of secure collaborative applications in their entirety. A single software architect can model all relevant interactions and corresponding authorization policies, which facilitates efficient and consistent solutions. The involved participants of a distributed application are then provided with premade components that just need to be installed and configured. Alternatively, each stakeholder may develop its own components, which allows for full control over business logic and associated permissions. This supports ad-hoc collaboration between different organizations, where developers on each side specify entry-based interfaces with well-defined behavior and grant corresponding access rights to trusted subjects. Finally, access control may also increase the reliability of applications within a single organization by offering protection from programming errors. Each component may be assigned a different system user, so that interactions among them need to be authorized. Thus, authorization policies can be used to enforce contracts on the valid usage of interfaces.

As each distributed application component hosts its own space that can easily communicate with any other space, flexible application architectures are supported, from classical client-server models to decentralized P2P networks. Various hybrid variants are also possible, including hierarchical models with super-peers that take over specific coordination tasks, while other interactions are directly handled in a P2P style. The included access control mechanism can be adjusted accordingly. Authorization naturally follows the P2P paradigm via separate policies for each space, although centralized policy management may also be enabled with an administration server that injects rule entries into remote policy containers of subordinate spaces. In addition, also the configured authentication mechanisms can be adapted to the application architecture. In some scenarios, a central identity provider for all involved peers may be sufficient, while cross-organizational workflows usually require a federated authentication mechanism, where user management is handled autonomously by each company. To avoid the need for trusted servers, each space could also use its own local identity provider, but this would require different credentials per communication partner, which might be difficult to

manage in practice. Furthermore, some functionality, like the dynamic lookup of remote spaces, may still require central components.

The presented space-based coordination and security architectures are applicable for a wide range of practically relevant scenarios that require flexible collaboration among distributed components. Different architectural styles can be mapped to this extended SBC paradigm, including SOA, workflows, publish/subscribe, hybrid clouds, mobile agents, and self-organizing P2P networks. Due to the decentralized approach, a large amount of participants can be supported. The decoupled nature of space-based computing ensures scalability, as implicit load balancing can be achieved when distributed peers or services with the same configuration compete for task entries from a shared space. Coordination logic, state information, and authorization policies can be easily migrated to different runtime instances, which may run on dedicated servers or virtual machines.

Although the Peer Model clearly provides a more sophisticated programming paradigm, both middleware technologies targeted in this thesis have their merits. XVSM is more suitable for data-driven ad-hoc collaboration, as entries can be directly accessed without the need for the installation of wirings. Applications following the client-server style with offered services that may access backend components can be realized with the Secure Service Space as well as the Secure Peer Space. For more complex coordination constraints, the Secure Peer Space is usually the logical choice. It provides better control over the involved coordination logic, which is directly managed by the middleware. This is especially beneficial in multitenant scenarios (e.g., agent- or cloud-based architectures) and when coordination logic has to be adapted during run time (e.g., dynamic workflows).

To enable usage in real-world scenarios, the access control models have to be incorporated into the full software lifecycle. This includes application design and implementation, as well as subsequent deployment and maintenance phases. The presented approach fosters an integrated development process for secure collaborative applications. Suitable authorization policies that fit the given coordination logic can already be defined at design time and automatically activated during deployment by injecting the respective rule entries. For added flexibility, some parameters (e.g., names of authorized users) may be configurable during installation. Afterwards, dynamic policy updates are possible at any time, triggered either automatically by the coordination logic or manually by administrators. The Secure Peer Space additionally supports model-driven security, as each instantiated peer can be linked to its respective design via the content of its meta containers, which can be translated into a graphical model that incorporates both coordination logic and authorization rules.

If necessary, it is also possible to separate application development and security administration. In XVSM, instead of using their own embedded spaces, processes could communicate via external cores with appropriate authorization policies. In the Secure Peer Space, peers may be injected into remote host peers that handle access control for them. Administrators still have to understand the basic coordination logic so that they are able to specify suitable permissions, but developers need not consider access control. This way, also non-secure modules can be incorporated into secure applications, although expressiveness is limited compared to the integrated development approach.

Another important aspect for the practical feasibility of the presented technology is its performance. Naturally, the enforcement of highly expressive policies induces an overhead compared to using simple ACLs or similar mechanisms. Scalability (REQ-17) is considered by supporting concurrent authorization checks (due to the usage of multiple core processors or wiring executor threads) and fast execution for simple rules (due to the support for wildcards and the efficiency of the internal query mechanisms). The performed benchmarks prove that basic authorization policies can be evaluated quickly even for containers with many entries, while complex policies still enable sufficient performance in most practical scenarios. This particularly applies to the Secure Peer Space, which uses a more efficient scope evaluation mechanism that is integrated with the guard's PMQ evaluation.

Extensibility is supported via modular runtime architectures (REQ-18) with several extension points. In both cases, the authentication mechanism is fully exchangeable in order to allow different kind of identity providers. By modifying the PDP logic or relevant algorithms (e.g., for rule combination in XVSM), the expressiveness of the policy language can be restricted or extended without affecting other parts of the middleware. Additional aspects or system peers may provide further security features. This modular approach also supports heterogeneous middleware runtimes with varying security features that are customized for specific situations. A limited feature set may be provided for embedded systems, while enterprise versions for Java or .NET offer maximal expressiveness. If these versions are able to exchange entries in a platform-independent way, also access control between different middleware runtimes will be possible, as all relevant communication is bootstrapped via regular entry operations.

Reusability (REQ-19) is facilitated via the suggested pattern-based development process, where complex applications may be built from secure coordination patterns that specify permissions for common coordination tasks in a configurable way. However, in order to realize the vision of a development process based on tool-supported stepwise refinement from generic basic patterns to composed ones to domain-specific logic and finally deployable application components, an extension of the pattern catalog as well as further research regarding the formalization of patterns is required.

The practical feasibility of the approach has been validated via several use case scenarios, which were realized by means of case studies and student projects. The obtained feedback has shown that the security concepts and related APIs are comprehensible even without extensive documentation. Performance and stability of the middleware runtimes were sufficient, although they are still research prototypes. However, there is still room for improvement regarding usability and scalability, which has to be examined in further case studies that target different types of complex applications. Another important factor is the creation of a suitable toolchain (including graphical modeling and simulation tools) that supports application developers and security administrators in all phases of the software lifecycle.

CHAPTER 10

Conclusion

The main contribution of this thesis is the extension of SBC middleware with suitable security concepts that enable its application in open distributed environments. To cover a wide range of data-driven architectures, different forms of space-based coordination have been examined, ranging from an extensible space for ad-hoc coordination (XVSM) to a framework for dynamically specified distributed workflows (Peer Model).

For both coordination models, an expressive and flexible access control model has been devised, thus fulfilling objective O1 of this thesis. It could be shown that expressive access control mechanisms inspired by existing security standards and related systems can be successfully applied to SBC middleware by defining fine-grained permissions for the access to entries in distributed space containers. Each of the developed access control models is specified by means of a policy language and a corresponding security architecture that addresses authentication of incoming requests, authorization of different kinds of space operations, and management of authorization rules. Several features are bootstrapped with already existing middleware functionality in order to reduce the complexity of the specification. Thus, a reasonable balance between expressiveness and comprehensibility could be found.

Following an ABAC approach, authorization rules consider the content of accessed entries, security attributes of the accessing subject, and additional context properties. As authorization rules specify constraints with the same query language as the coordination logic, permissions can be custom-tailored for specific interactions (coordination-aware authorization). On the other hand, coordination logic may be influenced by authorization constraints that transparently filter out inaccessible entries (authorization-aware coordination). This enables integration of coordination logic and security constraints without creating a strong dependency, as both can be specified independently from each other. Thus, separation of concerns is preserved.

A holistic approach is followed for specifying different kind of privileges. Unified authorization rules determine permissions for data access, service invocations, adaptation of coordination logic, and security administration itself. As rule entries are stored in

special containers, flexible policy administration is supported by own mechanisms. Due to the decentralized authentication and authorization mechanisms, the access control models are suitable for open distributed environments with dynamically changing coordination logic and security constraints. Authorization is checked at multiple levels, thus enabling defense in depth. This especially applies to the Secure Peer Space with its hierarchical peer structure, which can be used to model nested security domains with different administrators. The chained delegation feature supports complex constraints on participants of a workflow, while trust-based rules explicitly specify which kind of delegations are permitted. This enables secure ad-hoc collaboration in a distributed system with varying trust assumptions and no central policy.

To accomplish objective O2, these security features were integrated into the respective middleware runtime architectures while preserving scalability and modularity. Authorization modules reuse existing query functions to efficiently determine access decisions, while authentication mechanisms are fully exchangeable. Pre-existing prototype implementations of XVSM and the Peer Space have been extended accordingly to demonstrate the feasibility of the approach. Initial benchmarks demonstrate a reasonable overhead for access control that should be irrelevant for the majority of practical coordination problems. Depending on the scenario, alternative access control profiles may be defined to decrease the security overhead, as outlined by a security concept targeting a Peer Space version for embedded devices.

Regarding objective O3, it could be shown that pattern concepts for the specification of generic coordination logic can be enriched with corresponding authorization constraints. Thus, secure coordination patterns have been introduced, which depict reusable solutions that ensure effective and secure coordination for common scenarios. They provide configurable models of the coordination logic and accompanying authorization policies together with a textual explanation of the problem context, applied design decisions, and possible fields of application. The presented basic set of patterns, which was extracted from several use cases with complex coordination and security requirements, constitutes a starting point for the definition of a systematic pattern catalog and a novel software development approach based on pattern composition and refinement.

The presented approach has been evaluated using diverse methods. A comparison with related work shows that the described access control models support a more complete feature set than other approaches in the area of secure coordination middleware. General-purpose security frameworks used in today's distributed applications may provide similar features, but they are not geared to the special requirements of data-driven coordination and therefore treat the space as a black box, whereas an integrated access control mechanism provides a more flexible and expressive way of handling permissions. In addition, the effects of possible security vulnerabilities on the access control models and its implementations have been examined. The theoretical analysis has shown that all the defined requirements are appropriately addressed and that the access control extensions ensure a high level of security when applied correctly. Different application scenarios are conceivable, from server-centric applications to decentralized P2P networks with mutually mistrusting stakeholders. However, practical feasibility can only be proven via

realistic case studies. Therefore, secure solutions for the examined use cases have been designed. The specified pattern solution models were validated through implementation on top of the Secure Peer Space middleware combined with corresponding test cases. As demonstrated by the examples given within this thesis, complex constraints could be realized with relatively simple authorization rules without significantly affecting the coordination logic.

The key results of this thesis can be summarized as follows:

- Fine-grained access control with expressive rules is a necessary feature of coordination middleware in open distributed environments.
- Coordination and access control can be combined in a natural way by using the same query mechanisms for both.
- Operation requests should not only be authorized when first entering the system, but whenever they directly or indirectly trigger access to a space container. This provides multiple layers of protection.
- Advanced delegation mechanisms are useful to express complex trust relations within distributed workflows, although they are mostly ignored by current access control models for coordination middleware.
- Bootstrapped administration features support the dynamic and secure modification of coordination logic and permissions, thus enabling highly adaptive applications that can quickly react to changing requirements.
- Secure coordination patterns facilitate a new way of creating applications that are secure by design based on well-established authorization strategies for specific interactions.

10.1 Future Work

The focus of this thesis was on the development and evaluation of new access control concepts for SBC middleware. It does neither provide a complete specification for secure coordination middleware nor a detailed description of supporting tools. Additional research is necessary to further improve the presented security features and enable their application in real-world projects. Therefore, several open issues have to be addressed in future work:

- **Integration with new Peer Model features:** The Peer Model specification is still evolving. Additional features like XVSM-like query semantics [Küh16], a comprehensive transaction model [Küh17], and invariant assertions [KRE18] have been suggested as extensions to the core functionality. The access control model has to be adapted accordingly to such changes.

- **Improved policy language:** While the presented policy languages already support fine-grained constraints, some aspects may be improved to enhance expressiveness and/or comprehensibility. Additional features may include more sophisticated query possibilities on subject trees (e.g., inspired by tree-based query languages like XPath [W3C17]) or complex conditions involving multiple interdependent container queries. Simplified subject templates may be helpful in cases where peer owners are identical with their respective runtime users, e.g., "$(A$ **for** $(B$ **for** $C)$ **@** $B)$ **@** A". In order to simplify restricted delegation of administrative rights, structured policy containers (similar to XACML policy sets) or recursive restrictions based on the rule owner's own permissions are possible. Other conceivable extensions are the support of secure cross-peer conditions and cryptographically verified delegations.

- **Formalization:** The detailed semantics of the presented access control models have to be specified in a formal way, which would provide a complete reference for future implementations and enable verification of relevant security properties using model checking. This task relies on the formalization of the underlying coordination models, which is ongoing research.

- **Fully bootstrapped runtime architecture:** A minimal runtime kernel for XVSM and the Peer Space could be used to bootstrap basic middleware functionality, including access control. In this setting, autonomous middleware components (e.g., for authentication, authorization, coordination, transactions, or IO) communicate in a decoupled way via meta-level space containers, which provides clear runtime semantics and improves extensibility (possibly at the cost of performance). Permissions of pluggable extension modules can then be easily configured with the available access control mechanisms. An internal identity provider could be realized in such a way.

- **Improved runtime implementations:** The current middleware prototypes need to be revised in order to provide a fully protected security kernel, include omitted features, and optimize performance. The integration of state-of-the-art technologies for federated identity management, authentication, and encrypted communication channels is highly advisable. To reduce the required training period for developers, simplified APIs and extensive documentation are necessary. Another open task is the integration of the access control features with additional runtime implementations like the embedded version of the Peer Space.

- **Interoperability:** Heterogeneous middleware runtimes with distinct security profiles should be able to interact, which would be highly useful in IoT scenarios that involve devices with widely different computing power. Besides defining an interoperable communication protocol, also the access control mechanisms have to be adapted accordingly to ensure compatibility between different versions.

- **Toolchain support:** The practical applicability of the approach highly depends on the provided tools for the design, implementation, testing, deployment, and

management of distributed applications. The described access control mechanisms have to be considered for such tools. A graphical modeler should provide a suitable abstraction for defining authorization policies, while a corresponding monitoring tool may visualize authorized and denied operations. GUI-based dynamic administration of permissions should be supported, while simulation and model checking tools need to consider possible attack vectors.

- **Pattern-based development process:** The semantics of pattern refinement operations (e.g., composition) has to be fully specified to enable their usage within the outlined software development approach. As a starting point for developers, a well-structured pattern catalog needs to be designed and populated with relevant secure coordination patterns at different abstraction levels. This should include high-level patterns for securing classic coordination problems like consensus and replication.
- **High-level security features:** Additional functionality can be bootstrapped by specifying corresponding secure coordination patterns. Possible extensions are hierarchical RBAC support, privacy policies for forwarded data, and the management of distributed authorization rules based on high-level security policies.
- **Usability tests:** As the described middleware systems and their access control models follow a paradigm that differs significantly from typical software development approaches, extensive usability tests with a representative group of developers and administrators are necessary to analyze applicability in typical scenarios. Thus, the benefits and drawbacks of the suggested technology can be evaluated, which may lead to further improvements of the provided tools and/or the underlying coordination and security concepts.
- **Real-world use cases:** A conclusive evaluation of the presented approach requires its usage for large-scale real-world applications in different application domains. Key properties like development time, code complexity, maintenance effort, frequency of security breaches or policy misconfigurations, and performance can be measured and compared to similar projects that use conventional technologies.
- **Generalization:** While the presented access control models were specifically designed for XVSM and the Peer Model, respectively, the underlying concepts can also be generalized for other types of coordination middleware with similar expressiveness. This includes distinctive features like the chained delegation mechanism or the nested policy structure. The integration with emergent technologies for distributed applications like cloud computing and blockchain appears particularly interesting.

APPENDIX A

Syntax Specification

This chapter describes relevant syntax definitions for the specification of entries, space operations, and authorization rules according to the simplified space variants used for this thesis. In addition, the internal structure of relevant system properties and meta entries is defined. As a complete formal specification of the spaces and their security extensions is out of scope of this work, some details are not fully specified, which helps to keep this definition concise.

The syntax is defined using a notation that combines elements of BNF [NBB+60] and EBNF [ISO96]. Non-terminals are enclosed by angle brackets, while quotes mark terminals. To improve readability, concatenation does not require explicit commas and each rule has its own paragraph to avoid the usage of a termination symbol. The usual EBNF notation is used for alternatives, groupings, optional and repeated segments, as well as comments and special sequences that are not formally defined. To prevent repetition, rules may refer to non-terminals from earlier sections. Non-terminals may also be redefined in a later section to enable partial reuse.

A.1 General Specifications

This section contains syntax specifications that are relevant for both XVSM and the Peer Model.

A.1.1 Data Types

Overall data types used within subsequent definitions are defined in the following.

⟨*String*⟩ = "`" ⟨*CharSeq*⟩ "´"

⟨*CharSeq*⟩ = *? any valid sequence of visible characters ?*

⟨*Identifier*⟩ = *? valid ID consisting of letters, digits, and selected special characters ?*

⟨*Int*⟩ = *? any positive or negative integer value (including 0) ?*

⟨*PosInt*⟩ = *? any integer value > 0 ?*

⟨*PosInt0*⟩ = *? any integer value ≥ 0 ?*

⟨*Float*⟩ = *? any positive or negative float value (including 0.0) ?*

⟨*Bool*⟩ = `"true"` | `"false"`

⟨*Object*⟩ = *? valid representation of arbitrary object (or its reference) ?*

⟨*Domain*⟩ = *? valid representation of an Internet domain ?*

⟨*URL*⟩ = *? valid Internet address with scheme, host name, and (optionally) port ?*

⟨*SpaceURL*⟩ = ⟨*URL*⟩

⟨*UserID*⟩ = ⟨*Identifier*⟩

⟨*Type*⟩ = ⟨*Identifier*⟩

A.1.2 Entries

The following grammars define the general syntax of entries and specify how properties within them can be accessed via their path.

Entry Structure

⟨*Entry*⟩ = ⟨*PropertySet*⟩

⟨*PropertySet*⟩ = `"["` [⟨*Property*⟩ { `","` ⟨*Property*⟩ }] `"]"`

⟨*Property*⟩ = ⟨*PropKey*⟩ `":"` ⟨*PropVal*⟩

⟨*PropKey*⟩ = ⟨*Identifier*⟩

⟨*PropVal*⟩ = ⟨*Data*⟩ | ⟨*PropertySet*⟩ | ⟨*ValueList*⟩

⟨*ValueList*⟩ = `"⟨"` [⟨*PropVal*⟩ { `","` ⟨*PropVal*⟩ }] `"⟩"`

⟨*Data*⟩ = ⟨*String*⟩ | ⟨*Int*⟩ | ⟨*Bool*⟩ | ⟨*Float*⟩ | ⟨*Object*⟩ | ⟨*NullValue*⟩

⟨*NullValue*⟩ = `"NULL"` *(* represents an undefined property *)*

Property Selection

⟨*PropPath*⟩ = ⟨*SimplePropRef*⟩ { "." ⟨*SimplePropRef*⟩ }

⟨*SimplePropRef*⟩ = ⟨*PropKey*⟩ { "[" ⟨*PropIndex*⟩ "]" }

⟨*PropIndex*⟩ = ⟨*PosInt*⟩

A.1.3 Property Filters

The following definition defines expressive filters, which represent Boolean functions that test whether an entry fulfills specified criteria. The syntax is defined in a rather general way to allow for extensibility. Besides adhering to the given syntax, a valid filter must also use data types that are compatible with the corresponding operators.

⟨*PropFilter*⟩ = ⟨*Expr*⟩ ⟨*CompareOp*⟩ ⟨*Expr*⟩
| ⟨*Expr*⟩ ⟨*SetOp*⟩ ⟨*Expr*⟩
| ⟨*BoolFunc*⟩ "(" [⟨*Expr*⟩ { "," ⟨*Expr*⟩ }] ")"
| "¬" ⟨*PropFilter*⟩
| ⟨*PropFilter*⟩ "∧" ⟨*PropFilter*⟩
| ⟨*PropFilter*⟩ "∨" ⟨*PropFilter*⟩
| "(" ⟨*PropFilter*⟩ ")"

⟨*Expr*⟩ = ⟨*ExprValue*⟩
| ⟨*UnOp*⟩ ⟨*Expr*⟩
| ⟨*Expr*⟩ ⟨*BinOp*⟩ ⟨*Expr*⟩
| ⟨*Function*⟩ "(" [⟨*Expr*⟩ { "," ⟨*Expr*⟩ }] ")"
| "(" ⟨*Expr*⟩ ")"

⟨*ExprValue*⟩ = ⟨*PropVal*⟩ | ⟨*PropPath*⟩ | ⟨*ExprSet*⟩

⟨*ExprSet*⟩ = "{" [⟨*ExprValue*⟩ { "," ⟨*ExprValue*⟩ }] "}"

⟨*UnOp*⟩ = *? supported unary operators, e.g., negation ?*

⟨*BinOp*⟩ = *? supported binary operators, e.g., addition or concatenation ?*

⟨*Function*⟩ = *? predefined functions, e.g., size of a list or set ?*

⟨*CompareOp*⟩ = "=" | "≠" | "<" | "≤" | ">" | "≥"

⟨*SetOp*⟩ = "∈" | "∉" | "⊆"

⟨*BoolFunc*⟩ = *? predefined Boolean functions, e.g., existence of a property ?*

A.2 XVSM

This section contains relevant syntax definitions for XVSM and its access control model.

A.2.1 CAPI Function Parameters

The arguments and return values of the XVSM CAPI functions have already been specified in Table 5.1. In the following, the structure of the used data types is defined.

General

⟨*SpaceRef*⟩ = ⟨*SpaceURL*⟩ | ⟨*NullValue*⟩

⟨*ContainerRef*⟩ = [⟨*SpaceURL*⟩ "/"] ⟨*ContainerID*⟩

⟨*ContainerID*⟩ = ⟨*Identifier*⟩

⟨*TxID*⟩ = ⟨*Identifier*⟩ | ⟨*NullValue*⟩

⟨*OpTimeout*⟩ = ⟨*PosInt*⟩ | "INFINITE" | "TRY_ONCE" | "ZERO"

⟨*Context*⟩ = ⟨*PropertySet*⟩

⟨*Operation*⟩ = ⟨*SpaceOp*⟩ | ⟨*ContainerOp*⟩

⟨*SpaceOp*⟩ = "CreateContainer" | "DestroyContainer"
| "LookupContainer" | "CreateTransaction"
| "CommitTransaction" | "RollbackTransaction"
| "AddAspect" | "RemoveAspect"

⟨*ContainerOp*⟩ = "Write" | "Read" | "Take"

⟨*CoordName*⟩ = "any" | "fifo" | "lifo" | "vector" | "key" | "label"
| "type" | "linda" | "query" | ⟨*CustomCoordName*⟩

⟨*CustomCoordName*⟩ = ⟨*Identifier*⟩

⟨*VectorIndex*⟩ = ⟨*PosInt0*⟩

⟨*Key*⟩ = ⟨*String*⟩

⟨*Label*⟩ = ⟨*String*⟩

Write Operation

⟨*EntriesWithCoData*⟩ = "⟨" ⟨*EntryWithCoData*⟩ { "," ⟨*EntryWithCoData*⟩ } "⟩"

⟨*EntryWithCoData*⟩ = "(" ⟨*Entry*⟩ "," ⟨*CoData*⟩ ")"

⟨*CoData*⟩ = "[" [⟨*CoordRegistration*⟩ { "," ⟨*CoordRegistration*⟩ }] "]"

⟨*CoordRegistration*⟩ = ⟨*VectorReg*⟩ | ⟨*KeyReg*⟩ | ⟨*LabelReg*⟩ | ⟨*TypeReg*⟩
| ⟨*CustomCoordReg*⟩

⟨*VectorReg*⟩ = "vector(" ⟨*VectorInsertPos*⟩ ")"

⟨*VectorInsertPos*⟩ = ⟨*VectorIndex*⟩ | "APPEND"

⟨*KeyReg*⟩ = "key(" ⟨*Key*⟩ ")"

⟨*LabelReg*⟩ = "label(" ⟨*Label*⟩ { "," ⟨*Label*⟩ } ")"

⟨*TypeReg*⟩ = "type(" ⟨*Type*⟩ ")"

⟨*CustomCoordReg*⟩ = ⟨*CustomCoordName*⟩ "(" ⟨*RegArg*⟩ { "," ⟨*RegArg*⟩ } ")"

⟨*RegArg*⟩ = ⟨*Data*⟩

Query Operations

⟨*XQuery*⟩ = ⟨*Selector*⟩ { "|" ⟨*Selector*⟩ }

⟨*Selector*⟩ = ⟨*AnySelector*⟩ | ⟨*FifoSelector*⟩ | ⟨*LifoSelector*⟩ | ⟨*VectorSelector*⟩
| ⟨*KeySelector*⟩ | ⟨*LabelSelector*⟩ | ⟨*TypeSelector*⟩ | ⟨*LindaSelector*⟩
| ⟨*QuerySelector*⟩ | ⟨*CustomSelector*⟩

⟨*Count*⟩ = ⟨*PosInt*⟩ | "ALL" | "MAX"

⟨*AnySelector*⟩ = "any(" [⟨*Count*⟩] ")"

⟨*FifoSelector*⟩ = "fifo(" [⟨*Count*⟩] ")"

⟨*LifoSelector*⟩ = "lifo(" [⟨*Count*⟩] ")"

⟨*VectorSelector*⟩ = "vector(" ⟨*VectorParam*⟩ ["," ⟨*Count*⟩] ")"

⟨*VectorParam*⟩ = ⟨*VectorIndex*⟩

⟨*KeySelector*⟩ = "key(" ⟨*KeyParam*⟩ ")"

⟨*KeyParam*⟩ = ⟨*Key*⟩

⟨*LabelSelector*⟩ = "label(" ⟨*LabelParam*⟩ ["," ⟨*Count*⟩] ")"

⟨*LabelParam*⟩ = ⟨*Label*⟩

⟨*TypeSelector*⟩ = "type(" ⟨*TypeParam*⟩ ["," ⟨*Count*⟩] ")"

⟨*TypeParam*⟩ = ⟨*Type*⟩

⟨*LindaSelector*⟩ = "linda(" ⟨*Template*⟩ ["," ⟨*Count*⟩] ")"

⟨*Template*⟩ = "[" [⟨*TmplField*⟩ { "," ⟨*TmplField*⟩ }] "]"

⟨*TmplField*⟩ = ⟨*PropPath*⟩ ":" ⟨*TmplFieldVal*⟩

⟨*TmplFieldVal*⟩ = ⟨*PropVal*⟩ | "*"

⟨*QuerySelector*⟩ = "query(" ⟨*Query*⟩ { "|" ⟨*Query*⟩ } ")"

⟨*Query*⟩ = ⟨*PropFilter*⟩
| "sortup(" ⟨*PropPath*⟩ ")"
| "sortdown(" ⟨*PropPath*⟩ ")"
| "distinct(" ⟨*PropPath*⟩ ")"
| "reverse()"
| "cnt(" ⟨*MinCnt*⟩ "," ⟨*MaxCnt*⟩ ")"

⟨*MinCnt*⟩ = ⟨*PosInt0*⟩

⟨*MaxCnt*⟩ = ⟨*Count*⟩

⟨*CustomSelector*⟩ = ⟨*CustomCoordName*⟩ "(" [⟨*SelArg*⟩ { "," ⟨*SelArg*⟩ }] ")"

⟨*SelArg*⟩ = ⟨*Data*⟩

⟨*EntryList*⟩ = "⟨" [⟨*Entry*⟩ { "," ⟨*Entry*⟩ }] "⟩"

Management Operations

⟨*ContainerName*⟩ = ⟨*String*⟩ | ⟨*NullValue*⟩

⟨*ContainerSize*⟩ = ⟨*PosInt*⟩ | "UNBOUNDED"

⟨*CoordConfig*⟩ = "[" ⟨*CoordDef*⟩ { "," ⟨*CoordDef*⟩ } "]"

⟨*CoordDef*⟩ = ⟨*CoordName*⟩ "(" [⟨*InitArg*⟩ { "," ⟨*InitArg*⟩ }] ")"

⟨*InitArg*⟩ = ⟨*Data*⟩

⟨*TxTimeout*⟩ = ⟨*PosInt*⟩ | "INFINITE"

⟨*IPoint*⟩ = ("pre" | "post") ⟨*OpConstraint*⟩

⟨*OpConstraint*⟩ = ⟨*SpaceOp*⟩
| ⟨*ContainerOp*⟩ ["(" ⟨*ContainerID*⟩ ")"]

⟨*AspectID*⟩ = ⟨*Identifier*⟩

⟨*AspectImpl*⟩ = *? valid representation of aspect implementation ?*

A.2.2 Request Handling

Request and response entries used within the XVSM runtime architecture adhere to the following structure:

⟨*Request*⟩ = "[" ⟨*IDProp*⟩ ["," ⟨*CallerProp*⟩] "," ⟨*OpProp*⟩ "," ⟨*ReqArgs*⟩ "]"

⟨*IDProp*⟩ = "id:" "`" ⟨*Identifier*⟩ "'"

⟨*CallerProp*⟩ = "caller:" "`" ⟨*SpaceURL*⟩ "'"

⟨*OpProp*⟩ = "op:" "`" ⟨*Operation*⟩ "'"

⟨*ReqArgs*⟩ = ⟨*Property*⟩ { "," ⟨*Property*⟩ }

⟨*Response*⟩ = "[" ⟨*IDProp*⟩ ["," ⟨*CallerProp*⟩] "," ⟨*ResultProp*⟩ "]"

⟨*ResultProp*⟩ = "result:" ⟨*PropertySet*⟩

A.2.3 Security Context

The following definition extends the previously specified `Context` element to include security properties that are used for access control.

⟨*Context*⟩ = "[" ⟨*SubjectProp*⟩ ["," ⟨*CredProp*⟩] "," ⟨*RemoteFlag*⟩
["," ⟨*ForceAuthFlag*⟩] { "," ⟨*Property*⟩ } "]"

⟨*SubjectProp*⟩ = "subject:" ⟨*Subject*⟩

⟨*Subject*⟩ = "[" ⟨*SecurityAttr*⟩ { "," ⟨*SecurityAttr*⟩ } ["," ⟨*ExtraAttrs*⟩] "]"

⟨*SecurityAttr*⟩ = ⟨*UserProp*⟩ | ⟨*RoleProp*⟩ | ⟨*OrgProp*⟩ | ⟨*Property*⟩

⟨*UserProp*⟩ = "userId:" "`" ⟨*UserID*⟩ "'"

⟨*RoleProp*⟩ = "role:" ⟨*String*⟩

⟨*OrgProp*⟩ = "affiliation:" ⟨*String*⟩

⟨*ExtraAttrs*⟩ = "EA:" ⟨*PropertySet*⟩ *(* additional attributes, not authenticated *)*

⟨*CredProp*⟩ = "creds:" ⟨*Credentials*⟩ *(* removed after authentication *)*

⟨*Credentials*⟩ = ⟨*PropertySet*⟩

⟨*RemoteFlag*⟩ = "remoteRequest:" ⟨*Bool*⟩ *(* set by local runtime *)*

⟨*ForceAuthFlag*⟩ = "forceAuthorization:" ⟨*Bool*⟩ *(* default = false *)*

A.2.4 Policy Language

In the following, the syntax of XVSM authorization rules is specified, which reuses the XQuery element from the query operation parameters (see Section A.2.1). However, for adding support for dynamic parameters, some non-terminals referenced in the XQuery definition are refined here.

Rule Syntax

⟨*Rule*⟩ = "RULE" ⟨*RuleID*⟩
"SUBJECTS:" ("*" | (⟨*SubjTmpl*⟩ { "," ⟨*SubjTmpl*⟩ }))
"RESOURCES:" ("*" | (⟨*ContainerID*⟩ { "," ⟨*ContainerID*⟩ }))
"ACTIONS:" ("*" | (⟨*AccessMode*⟩ { "," ⟨*AccessMode*⟩ }))
"SCOPE:" ("*" | ⟨*Scope*⟩)
"CONDITION:" ("-" | ⟨*Condition*⟩)
"EFFECT:" ("PERMIT" | "DENY")

⟨*RuleID*⟩ = ⟨*Identifier*⟩

⟨*SubjTmpl*⟩ = "[" ⟨*PropPath*⟩ ":" ⟨*PropVal*⟩ { "," ⟨*PropPath*⟩ ":" ⟨*PropVal*⟩ } "]"

⟨*AccessMode*⟩ = "write" | "read" | "take"

⟨*Scope*⟩ = ⟨*XQuery*⟩
| "NOT" ⟨*Scope*⟩
| ⟨*Scope*⟩ "AND" ⟨*Scope*⟩
| ⟨*Scope*⟩ "OR" ⟨*Scope*⟩
| "(" ⟨*Scope*⟩ ")"

⟨*Condition*⟩ = ⟨*ContainerID*⟩ "|" ⟨*XQuery*⟩
| "NOT" ⟨*Condition*⟩
| ⟨*Condition*⟩ "AND" ⟨*Condition*⟩
| ⟨*Condition*⟩ "OR" ⟨*Condition*⟩
| "(" ⟨*Condition*⟩ ")"

Dynamic Parameter Integration

⟨*DynamicParam*⟩ = "$" ⟨*PropPath*⟩

⟨*Count*⟩ = ⟨*PosInt*⟩ | "ALL" | "MAX" | ⟨*DynamicParam*⟩

⟨*MinCnt*⟩ = ⟨*PosInt0*⟩ | ⟨*DynamicParam*⟩

⟨*VectorParam*⟩ = ⟨*VectorIndex*⟩ | ⟨*DynamicParam*⟩

⟨*KeyParam*⟩ = ⟨*Key*⟩ | ⟨*DynamicParam*⟩

⟨*LabelParam*⟩ = ⟨*Label*⟩ | ⟨*DynamicParam*⟩

⟨*TypeParam*⟩ = ⟨*Type*⟩ | ⟨*DynamicParam*⟩

⟨*TmplFieldVal*⟩ = ⟨*PropVal*⟩ | "⋆"
| ⟨*DynamicParam*⟩ *(* for linda selectors *)*

⟨*ExprValue*⟩ = ⟨*PropVal*⟩ | ⟨*PropPath*⟩ | ⟨*ExprSet*⟩
| ⟨*DynamicParam*⟩ *(* for query selectors *)*

⟨*SelArg*⟩ = ⟨*Data*⟩
| ⟨*DynamicParam*⟩ *(* for custom selectors *)*

A.2.5 Rule Entries

For the storage of rule entries in a policy container, the previously defined rule syntax is transformed into a specialized `Entry`. Wildcards are represented by omitting the respective properties.

⟨*RuleEntry*⟩ = "[" ⟨*RuleIdProp*⟩ ["," ⟨*SubjectsProp*⟩] ["," ⟨*ResourcesProp*⟩]
["," ⟨*ActionsProp*⟩] ["," ⟨*ScopeProp*⟩] ["," ⟨*ConditionProp*⟩]
"," ⟨*EffectProp*⟩ "]"

⟨*RuleIdProp*⟩ = "id:" "`" ⟨*RuleID*⟩ "'"

⟨*SubjectsProp*⟩ = "subjects:" "⟨" ⟨*SubjTmpl*⟩ { "," ⟨*SubjTmpl*⟩ } "⟩"

⟨*ResourcesProp*⟩ = "resources:" "⟨" ⟨*ContainerStr*⟩ { "," ⟨*ContainerStr*⟩ } "⟩"

⟨*ContainerStr*⟩ = "`" ⟨*ContainerID*⟩ "'"

⟨*ActionsProp*⟩ = "actions:" "⟨" ⟨*ModeStr*⟩ { "," ⟨*ModeStr*⟩ } "⟩"

⟨*ModeStr*⟩ = "`" ⟨*AccessMode*⟩ "'"

⟨*ScopeProp*⟩ = "scope:" "" ⟨*Scope*⟩ ""

⟨*ConditionProp*⟩ = "condition:" "" ⟨*Condition*⟩ ""

⟨*EffectProp*⟩ = "effect:" ("'PERMIT'" | "'DENY'")

A.3 Peer Model

This section addresses syntax definitions for the Peer Model and the Secure Peer Space.

A.3.1 Entry Structure

The definition of a Peer Model entry (PMEntry) extends the general Entry definition by providing a set of predefined properties.

⟨*PMEntry*⟩ = "[" ⟨*SystemCoProps*⟩ ["," ⟨*UserCoProps*⟩] ["," ⟨*AppData*⟩] "]"

⟨*SystemCoProps*⟩ = ⟨*EIDProp*⟩ "," ⟨*TypeProp*⟩ ["," ⟨*DESTProp*⟩]
["," ⟨*TTLProp*⟩] ["," ⟨*TTSProp*⟩] ["," ⟨*FlowProp*⟩]
{ "," ⟨*Property*⟩ }

⟨*EIDProp*⟩ = "EID:" "'" ⟨*EntryID*⟩ "'"

⟨*EntryID*⟩ = ⟨*Identifier*⟩

⟨*TypeProp*⟩ = "TYPE:" "'" ⟨*Type*⟩ "'"

⟨*DESTProp*⟩ = "DEST:" "'" ⟨*ContainerAddr*⟩ "'"

⟨*ContainerAddr*⟩ = [⟨*SpaceURL*⟩ "/"] ⟨*PeerAddr*⟩ ["/" ⟨*ContainerName*⟩]
| ⟨*SpaceURL*⟩ ["/" ⟨*ContainerName*⟩] *(* remote runtime peer *)*
| "/" ⟨*ContainerName*⟩ *(* local runtime peer *)*

⟨*PeerAddr*⟩ = ⟨*PeerName*⟩ { "/" ⟨*PeerName*⟩ }

⟨*PeerName*⟩ = ⟨*Identifier*⟩

⟨*ContainerName*⟩ = "PIC" | "POC" | "PSC" | "WSC" | "SPC"
| ⟨*Identifier*⟩ *(* using different namespace than PeerName *)*

⟨*TTLProp*⟩ = "TTL:" ⟨*Timer*⟩

⟨*TTSProp*⟩ = "TTS:" ⟨*Timer*⟩

⟨*Timer*⟩ = *? valid timer representation (absolute and/or relative time) ?*

⟨*FlowProp*⟩ = "`FID:`" "`\`" (⟨*Identifier*⟩ | "`NEW_FLOW`") "`'`"

⟨*UserCoProps*⟩ = ⟨*Property*⟩ { "`,`" ⟨*Property*⟩ }

⟨*AppData*⟩ = "`data:`" ⟨*Data*⟩

A.3.2 Peer Model Queries

The syntax of a PMQ, which is used for retrieving entries via a link, is largely based on the `PropFilter` definition from Section A.1.3.

⟨*PMQ*⟩ = ⟨*Type*⟩ ["`[`" ⟨*Count*⟩ "`]`"] ["⟦" ⟨*Selector*⟩ "⟧"]

⟨*Count*⟩ = (⟨*CntVal*⟩ "`;`" ⟨*CntVal*⟩) | ⟨*CntVal*⟩ | (⟨*RelOp*⟩ ⟨*CntVal*⟩) | "`ALL`"

⟨*CntVal*⟩ = ⟨*PosInt0*⟩

⟨*RelOp*⟩ = "<" | "≤" | ">" | "≥"

⟨*Selector*⟩ = ⟨*PropFilter*⟩

A.3.3 Link Assignments

Link assignments reuse expressions (`Expr`) as defined in Section A.1.3 for setting local variables and property values. Therefore, the `ExprValue` element is extended to also include variables.

⟨*AssignmentList*⟩ = "⟨" ⟨*Assignment*⟩ { "`,`" ⟨*Assignment*⟩ } "⟩"

⟨*Assignment*⟩ = ⟨*PropAssignment*⟩ | ⟨*VarAssignment*⟩

⟨*PropAssignment*⟩ = ⟨*PropPath*⟩ "=" ⟨*Expr*⟩

⟨*VarAssignment*⟩ = ⟨*Variable*⟩ "=" ⟨*Expr*⟩

⟨*Variable*⟩ = "`$`" ⟨*Identifier*⟩

⟨*SysVar*⟩ = "`$$`" ⟨*SysVarId*⟩ [⟨*SubPropRef*⟩]

⟨*SysVarId*⟩ = "`TIME`" | "`THIS_PEER`" | "`SELF`"
| ⟨*Identifier*⟩ *(* unspecified additional system variables *)*

⟨*SubPropRef*⟩ = "`.`" ⟨*PropPath*⟩ *(* for accessing fields of nested variables *)*

⟨*ExprValue*⟩ = ⟨*PropVal*⟩ | ⟨*PropPath*⟩ | ⟨*ExprSet*⟩
| ⟨*Variable*⟩ | ⟨*SysVar*⟩

A.3.4 Meta Model Entries

The following grammars show suitable structures for peer and wiring specification entries, which are specializations of `PMEntry` that contain meta information within their coordination properties.

Peer Specification Entries

⟨*PeerSpecEntry*⟩ = "[" ⟨*SystemCoProps*⟩ "," ⟨*PeerProps*⟩ "]"

⟨*TypeProp*⟩ = "TYPE:" "'Peer'"

⟨*PeerProps*⟩ = ⟨*PeerIDProp*⟩ { "," ⟨*Property*⟩ }

⟨*PeerIDProp*⟩ = "pid:" "`" ⟨*Identifier*⟩ "'"

Wiring Specification Entries

⟨*WiringSpecEntry*⟩ = "[" ⟨*SystemCoProps*⟩ "," ⟨*WiringProps*⟩ "]"

⟨*TypeProp*⟩ = "TYPE:" "'Wiring'"

⟨*WiringProps*⟩ = ⟨*WiringIDProp*⟩ "," ⟨*GuardsProp*⟩ "," ⟨*ActionsProp*⟩
["," ⟨*ServiceProp*⟩] ["," ⟨*RepeatProp*⟩] { "," ⟨*Property*⟩ }

⟨*WiringIDProp*⟩ = "wid:" "`" ⟨*Identifier*⟩ "'"

⟨*GuardsProp*⟩ = "guards:" "⟨" ⟨*GuardSpec*⟩ { "," ⟨*GuardSpec*⟩ } "⟩"

⟨*GuardSpec*⟩ = "[" ⟨*SourceProp*⟩ "," ⟨*LinkTypeProp*⟩ "," ⟨*PMQProp*⟩
["," ⟨*AssignmentsProp*⟩] "]"

⟨*SourceProp*⟩ = "source:" "`" ⟨*RelContainerAddr*⟩ "'"

⟨*RelContainerAddr*⟩ = ⟨*ContainerName*⟩
| ⟨*PeerName*⟩ "/" ⟨*ContainerName*⟩ *(* sub-peer container *)*

⟨*LinkTypeProp*⟩ = "linkType:" "`" ⟨*LinkType*⟩ "'"

⟨*LinkType*⟩ = "move" | "copy" | "test" | "delete" | "not"

⟨*PMQProp*⟩ = "query:" "[" ⟨*EntryTypeProp*⟩ ["," ⟨*CountProp*⟩]
["," ⟨*SelectorProp*⟩] "]"

⟨*EntryTypeProp*⟩ = "entryType:" "`" ⟨*Type*⟩ "'"

⟨*CountProp*⟩ = "count:" "`" ⟨*Count*⟩ "'"

⟨*SelectorProp*⟩ = "selector:" "'" ⟨*Selector*⟩ "'"

⟨*AssignmentsProp*⟩ = "assignments:" "⟨" ⟨*AssignDef*⟩ { "," ⟨*AssignDef*⟩ } "⟩"

⟨*AssignDef*⟩ = "'" ⟨*Assignment*⟩ "'"

⟨*ActionsProp*⟩ = "actions:" "⟨" [⟨*ActionSpec*⟩ { "," ⟨*ActionSpec*⟩ }] "⟩"

⟨*ActionSpec*⟩ = "[" ⟨*TargetProp*⟩ "," ⟨*LinkTypeProp*⟩ "," ⟨*PMQProp*⟩
["," ⟨*AssignmentsProp*⟩] "]"

⟨*TargetProp*⟩ = "target:" "`" ⟨*RelContainerAddr*⟩ "´"

⟨*ServiceProp*⟩ = "service:" "`" ⟨*ServiceRef*⟩ "´"

⟨*ServiceRef*⟩ = ⟨*Identifier*⟩

⟨*RepeatProp*⟩ = "repeat:" (⟨*PosInt*⟩ | "INFINITE")

A.3.5 Security Properties

For including access control features into the Peer Space, four additional system coordination properties have to be added to the previous `PMEntry` definition: `SubjectProp`, `CredProp`, `DelegationProp`, and `ImpersonateFlag`.

⟨*SubjectProp*⟩ = "SUBJECT:" ⟨*SubjectTree*⟩

⟨*SubjectTree*⟩ = "⟨" ⟨*ChildTree*⟩ { "," ⟨*ChildTree*⟩ } "⟩"

⟨*ChildTree*⟩ = "[" ⟨*PrincipalProps*⟩ ["," ⟨*ChildTreesProp*⟩] "]"

⟨*PrincipalProps*⟩ = ⟨*SecurityAttr*⟩ { "," ⟨*SecurityAttr*⟩ } "," ⟨*AuthContext*⟩

⟨*SecurityAttr*⟩ = ⟨*UserProp*⟩ | ⟨*DomainProp*⟩ | ⟨*RoleProp*⟩ | ⟨*OrgProp*⟩ | ⟨*Property*⟩

⟨*UserProp*⟩ = "userId:" "`" ⟨*UserID*⟩ "´"

⟨*DomainProp*⟩ = "domain:" "`" ⟨*Domain*⟩ "´"

⟨*RoleProp*⟩ = "role:" ⟨*String*⟩

⟨*OrgProp*⟩ = "org:" ⟨*String*⟩

⟨*AuthContext*⟩ = "authContext:" ⟨*PropertySet*⟩

⟨*ChildTreesProp*⟩ = "children:" ⟨*SubjectTree*⟩

⟨*CredProp*⟩ = "CREDS:" ⟨*Credentials*⟩ *(* removed after authentication *)*

⟨*Credentials*⟩ = "[" ⟨*ClaimedAttrs*⟩ "," ⟨*CredentialAttrs*⟩ "]"

⟨*ClaimedAttrs*⟩ = ⟨*SecurityAttr*⟩ { "," ⟨*SecurityAttr*⟩ }

⟨*CredentialAttrs*⟩ = ⟨*Property*⟩ { "," ⟨*Property*⟩ }

⟨*DelegationProp*⟩ = "DLG:" "`" ⟨*EntryID*⟩ "'" *(* optional *)*

⟨*ImpersonateFlag*⟩ = "IMP:" ⟨*Bool*⟩ *(* optional, default = false *)*

A.3.6 Subject Tree Representation

The following grammar defines the syntax for the textual subject tree notation.

⟨*SubjectTreeStr*⟩ = ⟨*NodeStr*⟩
| ⟨*SubjectTreeStr*⟩ "for" ⟨*SubjectTreeStr*⟩
| ⟨*SubjectTreeStr*⟩ "@" ⟨*NodeStr*⟩
| "(" ⟨*SubjectTreeStr*⟩ ")"

⟨*NodeStr*⟩ = "[" ⟨*PrincipalProps*⟩ "]"
| ⟨*UserID*⟩ *(* short for [userId: ‘* ⟨*UserID*⟩ *’] *)*

A.3.7 Policy Language

The syntax of Peer Model authorization rules is given in the following. It relies on query features from Section A.3.2. Support for context variables, system variables (see Section A.3.3), and aliases is added by redefining the `ExprValue` element from the property filter specification in Section A.1.3.

Rule Syntax

⟨*Rule*⟩ = "RULE" ⟨*RuleID*⟩
"SUBJECTS:" ⟨*SubjectTmpl*⟩ { "," ⟨*SubjectTmpl*⟩ }
"RESOURCES:" ("⋆" | (⟨*Container*⟩ { "," ⟨*Container*⟩ }))
"OPERATIONS:" ("⋆" | (⟨*AccessMode*⟩ { "," ⟨*AccessMode*⟩ }))
"SCOPE:" ("⋆" | ⟨*Scope*⟩)
"CONDITION:" ("-" | ⟨*Condition*⟩)

⟨*RuleID*⟩ = ⟨*Identifier*⟩

⟨*SubjectTmpl*⟩ = ⟨*NodeTmpl*⟩
| ⟨*SubjectTmpl*⟩ "`for`" ⟨*SubjectTmpl*⟩
| ⟨*SubjectTmpl*⟩ "`@`" ⟨*NodeTmpl*⟩
| "`(`" ⟨*SubjectTmpl*⟩ "`)`"

⟨*NodeTmpl*⟩ = "`*`" | "`**`" | ⟨*PrincipalTmpl*⟩

⟨*PrincipalTmpl*⟩ = "`[`" ⟨*Selector*⟩ { "`,`" ⟨*Selector*⟩ } "`]`"
| "`$$SELF`" *(* matches target peer owner *)*

⟨*Container*⟩ = ⟨*ContainerName*⟩ *(* when defined for specific peer *)*
| ⟨*LocalCAddr*⟩ *(* when defined at space level *)*

⟨*LocalCAddr*⟩ = ⟨*PeerAddr*⟩ "`/`" ⟨*ContainerName*⟩

⟨*AccessMode*⟩ = "`write`" | "`read`" | "`take`"

⟨*Scope*⟩ = ⟨*ScopeQuery*⟩
| "`NOT`" ⟨*Scope*⟩
| ⟨*Scope*⟩ "`AND`" ⟨*Scope*⟩
| ⟨*Scope*⟩ "`OR`" ⟨*Scope*⟩
| "`(`" ⟨*Scope*⟩ "`)`"

⟨*ScopeQuery*⟩ = ⟨*Type*⟩ ["`⟦`" ⟨*Selector*⟩ "`⟧`"]

⟨*Condition*⟩ = ⟨*ConditionQuery*⟩
| "`NOT`" ⟨*Condition*⟩
| ⟨*Condition*⟩ "`AND`" ⟨*Condition*⟩
| ⟨*Condition*⟩ "`OR`" ⟨*Condition*⟩
| "`(`" ⟨*Condition*⟩ "`)`"

⟨*ConditionQuery*⟩ = ⟨*Container*⟩ "`|`" ⟨*PMQ*⟩

Dynamic Parameter Integration

⟨*ContextVar*⟩ = "`$`" ⟨*PropPath*⟩

⟨*AliasPath*⟩ = ⟨*AliasId*⟩ ["`.`" ⟨*PropPath*⟩]

⟨*AliasId*⟩ = "`originator`" | "`invoker`" | "`sender`"
| ⟨*Identifier*⟩ *(* unspecified additional aliases *)*

⟨*AliasVar*⟩ = "`$`" ⟨*AliasPath*⟩

⟨*ExprValue*⟩ = ⟨*PropVal*⟩ | ⟨*PropPath*⟩ | ⟨*ExprSet*⟩
| ⟨*ContextVar*⟩ | ⟨*AliasPath*⟩ | ⟨*AliasVar*⟩ | ⟨*SysVar*⟩

A.3.8 Rule Entries

A rule entry is another specialization of `PMEntry` that stores rule data within its coordination properties. As in XVSM, wildcards are represented by the omission of optional properties.

⟨*RuleEntry*⟩ = "`[`" ⟨*SystemCoProps*⟩ "`,`" ⟨*RuleProps*⟩ "`]`"

⟨*TypeProp*⟩ = "`TYPE:`" "`'Rule'`"

⟨*RuleProps*⟩ = ⟨*RuleIdProp*⟩ "`,`" ⟨*SubjectsProp*⟩ ["`,`" ⟨*ResourcesProp*⟩] ["`,`" ⟨*OpsProp*⟩] ["`,`" ⟨*ScopeProp*⟩] ["`,`" ⟨*ConditionProp*⟩]

⟨*RuleIdProp*⟩ = "`rid:`" "`'`" ⟨*RuleID*⟩ "`'`"

⟨*SubjectsProp*⟩ = "`subjects:`" ⟨*SubjectTmplList*⟩

⟨*SubjectTmplList*⟩ = "⟨" ⟨*SubjTmplTree*⟩ { "`,`" ⟨*SubjTmplTree*⟩ } "⟩"

⟨*SubjTmplTree*⟩ = "⟨" ⟨*ChildTmplTree*⟩ { "`,`" ⟨*ChildTmplTree*⟩ } "⟩"

⟨*ChildTmplTree*⟩ = "`[`" ⟨*NodeTypeProp*⟩ ["`,`" ⟨*PrincipalTmplProp*⟩] ["`,`" ⟨*ChildTmplsProp*⟩] "`]`"

⟨*NodeTypeProp*⟩ = "`nodeType:`" "`'`" ⟨*NodeType*⟩ "`'`"

⟨*NodeType*⟩ = "`PRINCIPAL`" | "`*`" | "`**`" | "`SELF`"

⟨*PrincipalTmplProp*⟩ = "`constraints:`" "⟨" ⟨*SelectorDef*⟩ { "`,`" ⟨*SelectorDef*⟩ } "⟩"

⟨*SelectorDef*⟩ = "`"`" ⟨*Selector*⟩ "`"`"

⟨*ChildTmplsProp*⟩ = "`children:`" ⟨*SubjTmplTree*⟩

⟨*ResourcesProp*⟩ = "`resources:`" ⟨*ContainerList*⟩

⟨*ContainerList*⟩ = "⟨" ⟨*ContainerStr*⟩ { "`,`" ⟨*ContainerStr*⟩ } "⟩"

⟨*ContainerStr*⟩ = "`'`" ⟨*ContainerName*⟩ "`'`"

⟨*OpsProp*⟩ = "`operations:`" ⟨*OpList*⟩

⟨*OpList*⟩ = "⟨" ⟨*ModeStr*⟩ { "`,`" ⟨*ModeStr*⟩ } "⟩"

⟨*ModeStr*⟩ = "`'`" ⟨*AccessMode*⟩ "`'`"

⟨*ScopeProp*⟩ = "`scope:`" "`"`" ⟨*Scope*⟩ "`"`"

⟨*ConditionProp*⟩ = "`condition:`" "`"`" ⟨*Condition*⟩ "`"`"

APPENDIX B

Algorithms

This chapter describes selected algorithms that are relevant for the introduced access control models. For that purpose, UML activity diagrams are used.

B.1 XVSM Combination Algorithms

The configurable combination algorithm is part of the XVSM authorization workflow that is executed by the access manager. Using a set of rules with matching target that were retrieved from the policy container, it is responsible for computing a corresponding authorization result. Two different elements may be returned: a general decision object and/or a mapping of entries to individual decisions. If both elements are set, the general decision only applies to entries that are not covered by the mapping. If no general decision is defined, the default value for unspecified entries is `NOT_APPLICABLE`.

Depending on the followed strategy, conflicts involving multiple rules are handled differently. Figure B.1 shows a possible implementation for the `DENY-OVERRIDES` combination algorithm, where `DENY` rules take precedence over `PERMIT` rules. The activity diagram depicts the control flow as well as relevant changes to local variables.

B.2 Subject Template Matching

In the Peer Model, authorization rules contain subject templates that are matched with the responsible subject for the current access attempt. Both subjects and their templates are represented by tree data structures that indicate the involved principals and their relations. Figure B.2 outlines how this subject template matching is performed. For the sake of simplicity, it is assumed that the algorithm operates on delegation chains (i.e., lists of tree leaves) and authentication chains (i.e., lists of ancestor nodes for each element of a delegation chain), which can be easily extracted from both trees. The algorithm relies on the iterative traversal of these lists and the matching of corresponding nodes in

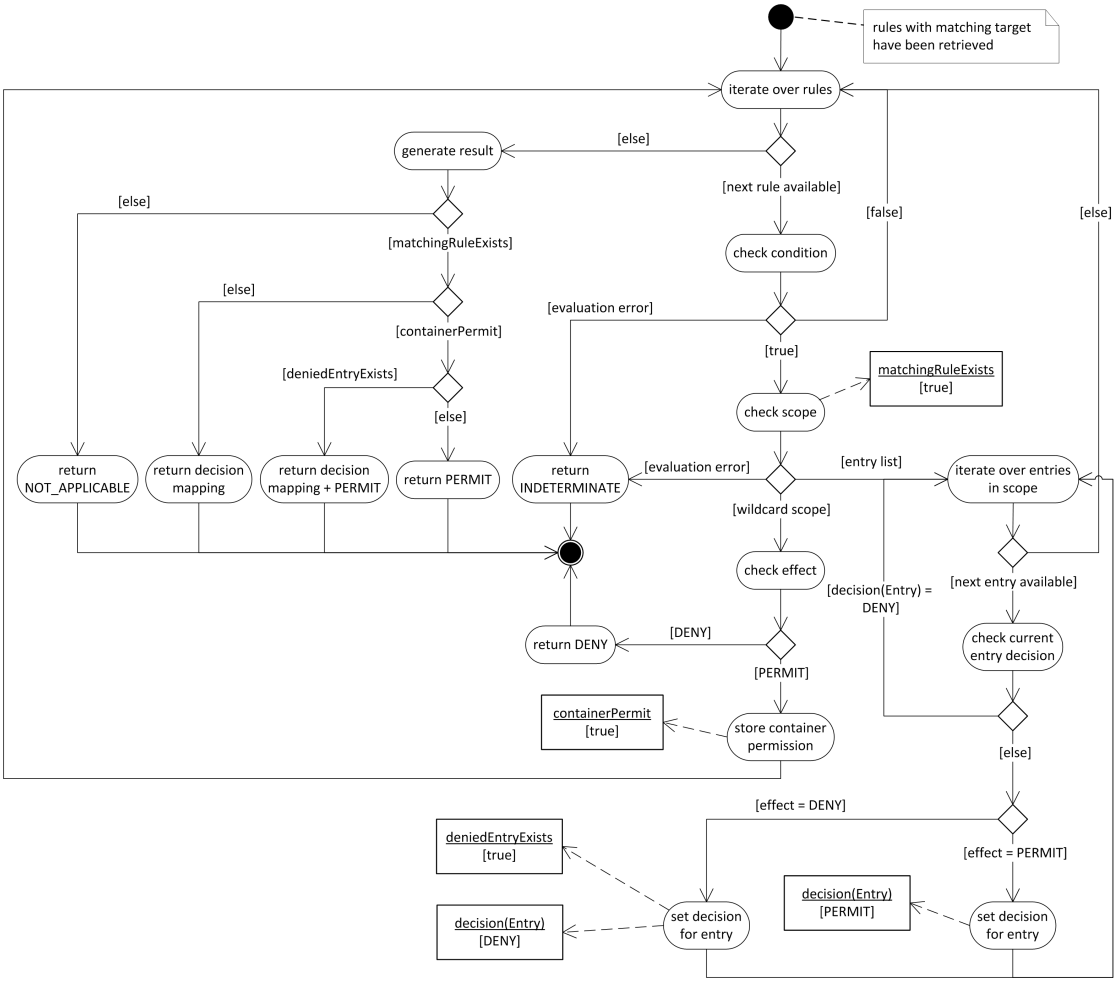

Figure B.1: Combination algorithm following `DENY-OVERRIDES` strategy

the subject and the subject template. A recursive call to the algorithm is required for the evaluation of "**" wildcards, which is represented by the composite "match template" activity. When a match occurs in the delegation chain, the respective authentication chains (without leaf and root nodes) are subsequently compared via the composite "match authentication chains" activity, which is shown in Figure B.3. This algorithm follows a similar strategy and also uses recursion for wildcard handling.

B.3 Secure Entry Routing

The destination mechanism of the Peer Model allows the routing of entries across peer and space boundaries. Access control mechanisms have to check permissions for all traversed containers. Figure B.4 shows the corresponding algorithm for entries with set `DEST` property that were emitted by local wiring actions. If an entry is injected from

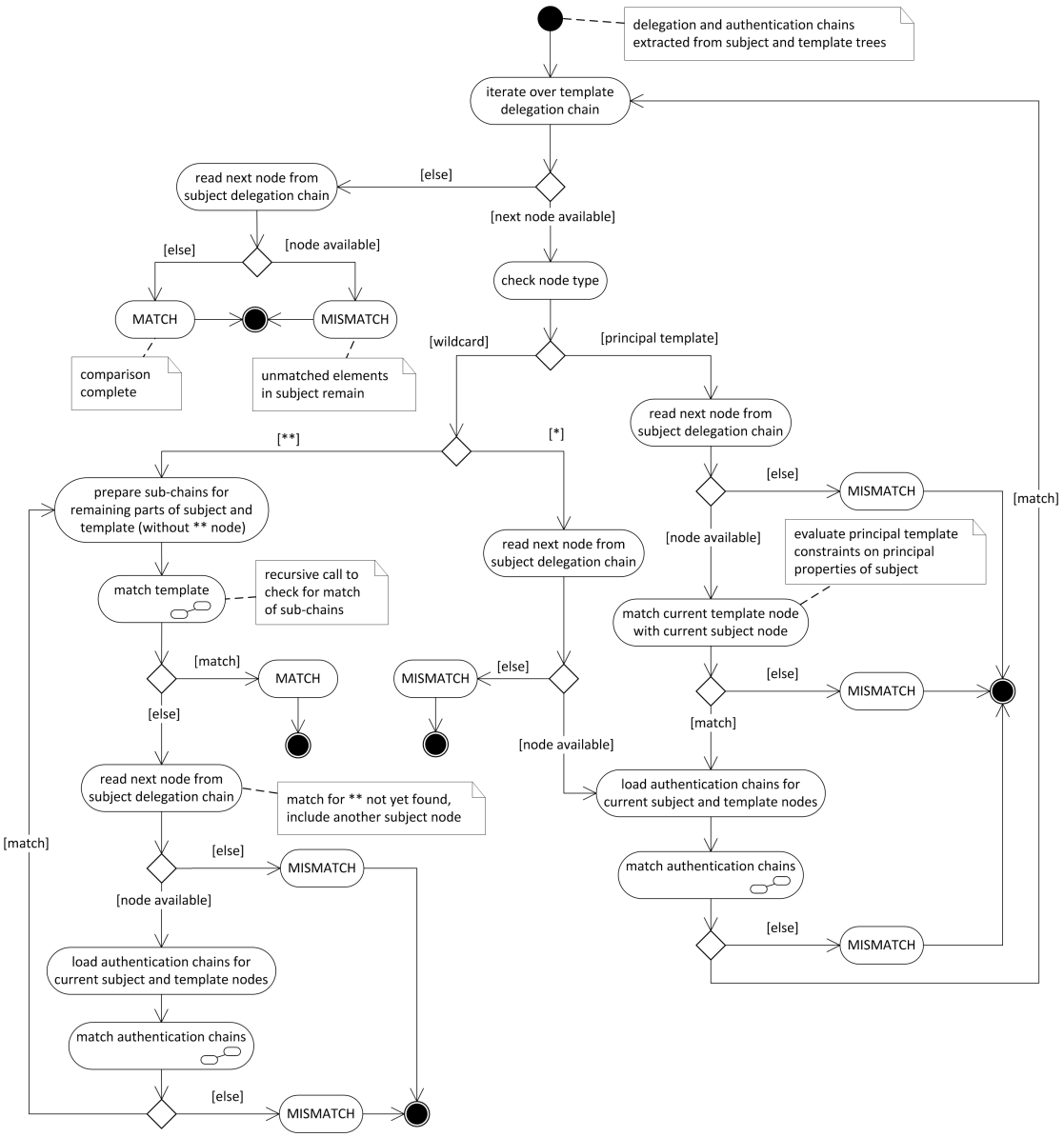

Figure B.2: Algorithm for matching subjects with subject templates

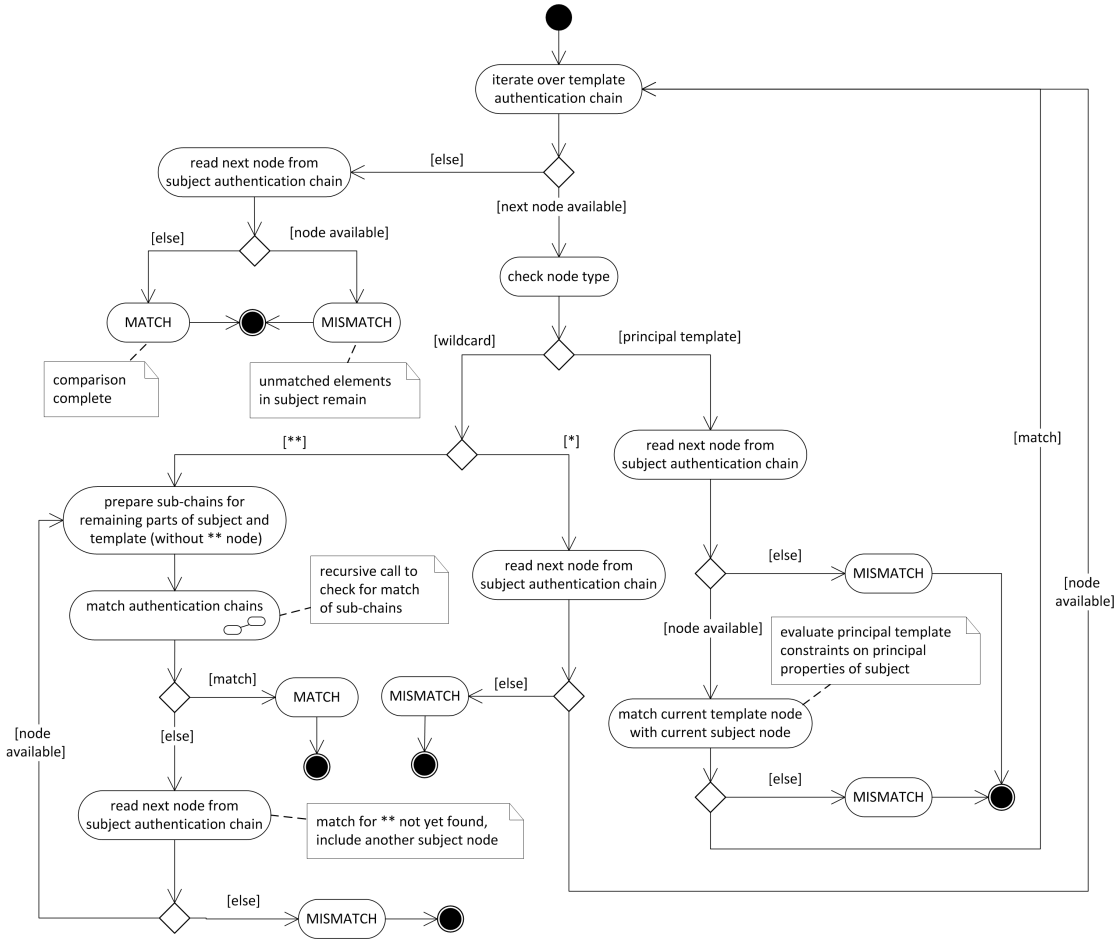

Figure B.3: Subroutine for matching authentication chains

a remote space or via the API, the second algorithm (Figure B.5) is applied. In both cases, error handling (e.g., for non-existing destination peers) is omitted for the sake of simplicity.

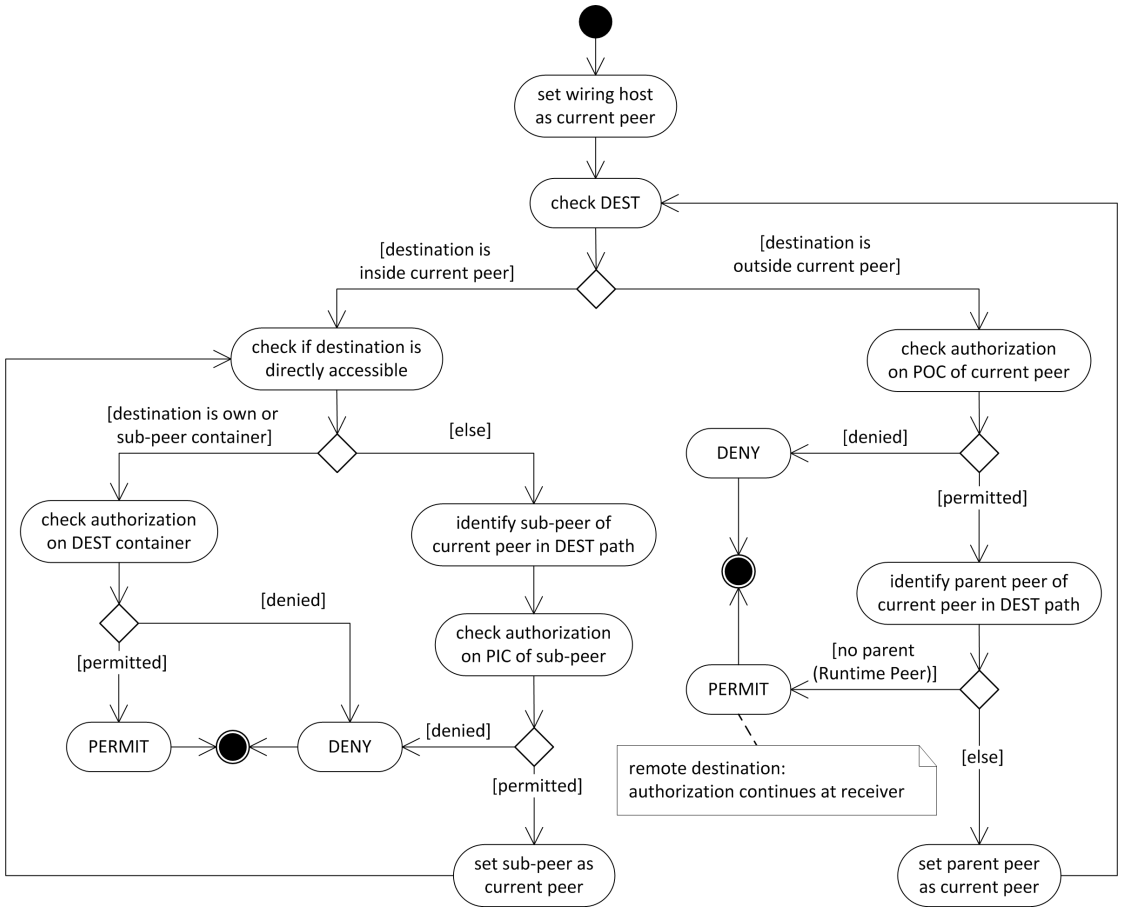

Figure B.4: Authorization checks for DEST routing (triggered by local wiring)

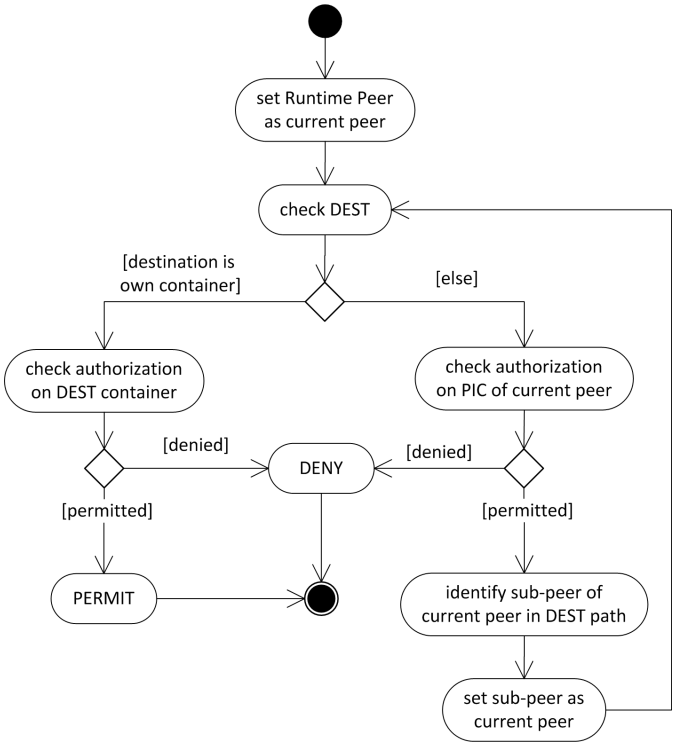

Figure B.5: Authorization checks for entry injection (via API or remote core)

List of Figures

List of Tables

List of Listings

Acronyms

2PC Two-phase commit. 169

AA Attribute authority. 19, 22

ABAC Attribute-based access control. 16, 17, 20, 22, 28, 38, 45, 51, 74, 115, 193, 197, 205

ACL Access control list. 14–16, 20–22, 24, 26, 28, 30, 40, 47, 63, 72, 78, 83, 102, 188, 195, 204

AES Advanced Encryption Standard. 137

API Application programming interface. 4, 5, 30, 32, 54, 55, 67, 68, 95, 103, 113, 114, 123, 130, 143, 151, 153, 164, 204, 208, 230, 232, 234, 235

AW-RBAC Adaptive Workflow RBAC. 35, 44, 45, 47, 51

BNF Backus-Naur form. 211

BPEL Business Process Execution Language. 4, 32, 36, 37, 39, 43

BPMN Business Process Model and Notation. 31, 32, 37–39, 43

CA Certificate authority. 19, 30, 142

CAEM Constraint Analysis and Enforcement Module. 34, 44, 47, 48

CAPI Core API. 68, 69, 71–74, 76, 79, 81–83, 88–92, 95, 115, 194, 214

CBSE Component-based software engineering. 31

CORBA Common Object Request Broker Architecture. 39, 40, 43, 49, 51

CORBAsec CORBA Security Service. 40, 44–48, 51

CPN Colored Petri net. 31, 32, 108

CPU Central processing unit. 57, 62, 64, 99, 102

CRL Certificate revocation list. 19

DA Delegated attributes. 92, 93

DAC Discretionary access control. 15, 20, 197

DEST Destination. 107, 109, 111–114, 134, 136, 152, 156, 228, 231, 234

DLG Delegation. 132, 154, 156

DSL Domain-specific language. 4, 5, 31, 32, 54, 55, 106, 114, 136, 137

DSSA Distributed System Security Architecture. 16, 20, 22, 48, 51, 197

EA Extra attributes. 85, 93

EBNF Extended Backus-Naur form. 54, 75, 121, 211

EID Entry ID. 107

ESB Enterprise service bus. 3

EWFN Executable Workflow Networks. 32

FID Flow ID. 107, 109, 177

FIFO First-in, first-out. 4, 23, 29, 31, 67, 70

GUI Graphical user interface. 34, 47, 60, 109, 209

HTTP Hypertext Transfer Protocol. 13, 17

IdP Identity provider. 117

IMP Impersonation. 132, 154

IO Input/output. 208

IoT Internet of things. 1, 208

ITU-T International Telecommunication Union — Telecommunication Standardization Sector. 18

JAAS Java Authentication and Authorization Service. 21

JDK Java Development Kit. 99

JIT Just-in-time. 99

JMS Java Message Service. 3, 39, 40

JVM Java Virtual Machine. 99, 143

KDC Key distribution center. 19, 20

KLAIM Kernel Language for Agents Interaction and Mobility. 25, 43–49

Lacios Language for Agent Contextual Interaction in Open Systems. 28, 43–48

LDAP Lightweight Directory Access Protocol. 13, 98

LGI Law-Governed Interaction. 27, 43–46, 48, 50, 51, 196

LGL Law-Governed Linda. 26, 43–46, 48, 196

LIFO Last-in, first-out. 70

Lime Linda in a Mobile Environment. 26, 37, 43–46, 49, 89, 173

LOPONODE Low-Power Node. 9, 60–62, 106, 135, 137, 181, 197, 233

MAC Message authentication code. 137, 138

MAC Mandatory access control. 15, 20, 39, 40

MDD Model-driven development. 31, 33

OASIS Organization for the Advancement of Structured Information Standards. 16, 17, 41

OpenAM Open Access Management. 21, 97

ORB Object Request Broker. 39, 40

OS Operating system. 201

P2P Peer-to-peer. 2, 6, 21, 37, 41, 42, 46, 62, 64, 67, 105, 117, 130, 155, 169, 174, 190, 191, 195, 202, 203

PAP Policy Administration Point. 18, 84, 131, 198

PDP Policy Decision Point. 18, 20, 30, 36, 38, 84, 130, 198, 204

PEP Policy Enforcement Point. 17, 18, 29, 30, 36, 84, 131, 134, 198

PERMIS PrivilEge and Role Management Infrastructure Standards. 16, 20, 22, 51, 197

PIC Peer-In-Container. 106, 107, 109, 111, 113, 115, 124–128, 134, 136, 143–145, 152–154, 156, 158, 159, 162, 164–167, 171, 175, 178, 188, 189

PIP Policy Information Point. 18, 84, 131, 198

PKI Public key infrastructure. 14, 17–21, 23, 29, 30, 51, 142

PMQ Peer Model Query. 108, 109, 111, 112, 114, 121–123, 125, 133–135, 139, 150, 151, 158, 173, 187, 195, 204, 221

POC Peer-Out-Container. 106, 109, 115, 126–129, 134, 136, 148, 154–156, 159, 160, 171, 172, 175, 177, 187

PSC Peer Specification Container. 111, 112, 114, 128, 129, 161, 170–172

QoS Quality of service. 60

RAM Random-access memory. 99

RBAC Role-based access control. 16, 17, 20, 22, 26, 33–36, 38, 39, 41, 45, 46, 51, 60, 190, 197, 209

RDF Resource Description Framework. 29

RMI Remote Method Invocation. 2

RPC Remote procedure call. 2, 39, 43, 155

RTP Runtime Peer. 111, 114, 125, 128, 131, 145–148, 153–156, 175, 177, 178, 186, 187

SAML Security Assertion Markup Language. 13, 14, 16, 17, 22, 29, 30, 39, 48, 194, 202

SASL Simple Authentication and Security Layer. 14, 202

SBC Space-based computing. 4, 7–9, 11, 16, 21–23, 29, 50, 54, 65, 103, 105, 106, 114, 124, 203, 205, 207

SecOS Secure Object Space. 24, 44, 49

SMC Security Management Center. 56–59, 64, 158, 166, 179, 181–185, 233, 234

SMEPP Secure Middleware for Embedded Peer-to-Peer systems. 37, 43–46, 49, 89

SOA Service-oriented architecture. 31, 203

SOAP Simple Object Access Protocol. 17, 21

SPC Security Policy Container. 126–129, 133, 134, 139, 147, 148, 161–163, 171, 172, 188, 189, 200

SPKI Simple public key infrastructure. 21, 41

SQL Structured Query Language. 4, 29, 34, 40, 43, 70

SSO Single sign-on. 13, 17, 20, 39, 51, 81, 97, 98, 131

T–RBAC Task–Role-Based Access Control. 34, 35, 39, 44, 45, 165

TAM Trust and Attribute Mapping. 29, 30

TCB Trusted computing base. 12, 103, 199, 202

TCP Transmission Control Protocol. 98

TGS Ticket-granting service. 20

TGT Ticket-granting ticket. 20

TLS Transport Layer Security. 17, 18, 29, 38, 142, 202

TSC Triple Space Computing. 29

TTL Time-to-live. 107, 108, 110, 112, 114, 129, 159, 161, 163, 200, 201

TTS Time-to-start. 107, 108, 112, 114, 129

TuCSoN Tuple Centres Spread over Networks. 26, 27, 32, 43, 44, 46–49, 51, 157, 196

UML Unified Modeling Language. 31, 32, 36, 43, 46, 54, 105, 227

URI Uniform Resource Identifier. 17, 69, 73, 107, 116, 123, 199

URL Uniform Resource Locator. 21, 111

VM Virtual machine. 99

VPN Virtual private network. 57, 182

WAM Workflow Authorization Model. 33, 34, 44, 45, 47, 48

WfMS Workflow management system. 32, 34–36, 39, 49

WIDE Workflow on Intelligent Distributed database Environment. 35, 43–45, 48, 50, 51

WS Web Services. 21, 38

WSC Wiring Specification Container. 111–115, 127, 139, 158–161, 170–172, 177

WSN Wireless sensor network. 49, 60, 61, 135, 138, 173, 181

XACML EXtensible Access Control Markup Language. 15–18, 22, 30, 36–38, 42, 45, 46, 48, 51, 74, 75, 79, 80, 84, 195, 198

XAP eXtreme Application Platform. 29, 43–47, 49, 51, 196

XML Extensible Markup Language. 16, 17, 20, 32, 36

XVSM EXtensible Virtual Shared Memory. 4, 5, 8, 9, 23, 24, 51, 53–56, 62, 63, 67, 68, 70–77, 79, 80, 82–84, 86, 87, 89, 91–93, 95–98, 102, 103, 105–108, 113–116, 121–125, 129, 130, 132, 143, 147, 150, 158, 160, 169, 181, 182, 185, 190, 191, 193–201, 203–209, 211, 214, 217, 218, 226, 227, 233–235

Bibliography

[Aal98] W.M.P. van der Aalst. The application of Petri nets to workflow management. *Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

[ABBD05] M. Altunay, D. Brown, G. Byrd, and R. Dean. Trust-based secure workflow path construction. In *Service-Oriented Computing — ICSOC 2005*, volume 3826 of *LNCS*, pages 382–395. Springer, 2005.

[ABF08] Eduardo A. P. Alchieri, Alysson Neves Bessani, and Joni da Silva Fraga. A dependable infrastructure for cooperative web services coordination. In *IEEE International Conference on Web Services (ICWS '08)*, pages 21–28. IEEE, 2008.

[ABLP93] Martín Abadi, Michael Burrows, Butler Lampson, and Gordon Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, 1993.

[AC93] Gul Agha and Christian J. Callsen. ActorSpace: an open distributed programming paradigm. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP '93)*, pages 23–32. ACM, 1993.

[ACC08] Samiha Ayed, Nora Cuppens-Boulahia, and Frédéric Cuppens. Deploying access control in distributed workflow. In *Proceedings of the Sixth Australasian Information Security Conference (AISC 2008)*, volume 81 of *Conferences in Research and Practice in Information Technology*, pages 9–17. Australian Computer Society, 2008.

[AH96] Vijayalakshmi Atluri and Wei-Kuang Huang. An authorization model for workflows. In *Computer Security — ESORICS 96*, volume 1146 of *LNCS*, pages 44–64. Springer, 1996.

[AH00] Vijayalakshmi Atluri and Wei-Kuang Huang. A Petri net based safety analysis of workflow authorization models. *Journal of Computer Security*, 8(2,3):209–240, 2000.

[AHB06a] Muhammad Alam, Michael Hafner, and Ruth Breu. Constraint based role based access control (CRBAC) for restricted administrative delegation constraints in the SECTET. In *Proceedings of the 2006 International Conference on Privacy, Security and Trust: Bridge the Gap Between PST Technologies and Business Services (PST '06)*, pages 44:1–5. ACM, 2006.

[AHB06b] Muhammad Alam, Michael Hafner, and Ruth Breu. A constraint based role based access control in the SECTET: A model-driven approach. In *Proceedings of the 2006 International Conference on Privacy, Security and Trust: Bridge the Gap Between PST Technologies and Business Services (PST '06)*, pages 13:1–13. ACM, 2006.

[AHBU06] Muhammad Alam, Michael Hafner, Ruth Breu, and Stefan Unterthiner. A framework for modeling restricted delegation in service oriented architecture. In *TrustBus 2006: Trust and Privacy in Digital Business*, volume 4083 of *LNCS*, pages 142–151. Springer, 2006.

[AHKB03] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[AIS77] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.

[AL03] Carlisle Adams and Steve Lloyd. *Understanding PKI: Concepts, Standards, and Deployment Considerations*. Addison-Wesley, 2nd edition, 2003.

[AM03] Xuhui Ao and Naftaly H. Minsky. Flexible regulation of distributed coalitions. In *Computer Security — ESORICS 2003*, volume 2808 of *LNCS*, pages 39–60. Springer, 2003.

[And04] David P. Anderson. BOINC: a system for public-resource computing and storage. In *Fifth IEEE/ACM International Workshop on Grid Computing*, pages 4–10. IEEE, 2004.

[Arb04] Farhad Arbab. Reo: A channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.

[ARDS15] Khalid Alissa, Jason Reid, Ed Dawson, and Farzad Salim. BP-XACML an authorisation policy language for business processes. In *Information Security and Privacy (ACISP 2015)*, volume 9144 of *LNCS*, pages 307–325. Springer, 2015.

[AT10] Tanvir Ahmed and Anand R. Tripathi. Security Policies in Distributed CSCW and Workflow Systems. *IEEE Transactions on Systems, Man, and Cybernetics — Part A: Systems and Humans*, 40(6):1220–1231, 2010.

[AW05] Vijayalakshmi Atluri and Janice Warner. Supporting conditional delegation in secure workflow management systems. In *Proceedings of the 10th ACM Symposium on Access Control Models and Technologies (SACMAT' 05)*, pages 49–58. ACM, 2005.

[AY16] Riham Altawy and Amr M. Youssef. Security tradeoffs in cyber physical systems: A case study survey on implantable medical devices. *IEEE Access*, 4:959–979, 2016.

[BACF08] Alysson Neves Bessani, Eduardo Pelison Alchieri, Miguel Correia, and Joni da Silva Fraga. DepSpace: A Byzantine fault-tolerant coordination service. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems (Eurosys '08)*, pages 163–176. ACM, 2008.

[Bar10] Martin-Stefan Barisits. Design and implementation of the next generation XVSM framework — operations, coordination and transactions. Master's thesis, TU Wien, 2010.

[BBB+08] F. Benigni, A. Brogi, J.L. Buchholz, J.M. Jacquet, J. Lange, and R. Popescu. Secure P2P Programming on Top of Tuple Spaces. In *17th Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 54–59. IEEE, 2008.

[BC01] Ciarán Bryce and Marco Cremonini. Coordination and security on the Internet. In A. Omicini, F. Zambonelli, M. Klusch, and R. Tolksdorf, editors, *Coordination of Internet Agents: Models, Technologies, and Applications*, pages 274–298. Springer, 2001.

[BCFL09] Alysson Neves Bessani, Miguel Correia, Joni da Silva Fraga, and Lau Cheuk Lung. Sharing memory between Byzantine processes using policy-enforced tuple spaces. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):419–432, 2009.

[BCP06] Elisa Bertino, Jason Crampton, and Federica Paci. Access control and authorization constraints for WS-BPEL. In *2006 IEEE International Conference on Web Services (ICWS '06)*, pages 275–284. IEEE, 2006.

[BDL03] David Basin, Jürgen Doser, and Torsten Lodderstedt. Model driven security for process-oriented systems. In *Proceedings of the 8th ACM Symposium on Access Control Models and Technologies (SACMAT '03)*, pages 100–109. ACM, 2003.

[BEP+03] András Belokosztolszki, David M. Eyers, Peter R. Pietzuch, Jean Bacon, and Ken Moody. Role-based access control for publish/subscribe middleware architectures. In *Proceedings of the 2nd International Workshop on Distributed Event-Based Systems (DEBS '03)*, pages 1–8. ACM, 2003.

[BFA99] Elisa Bertino, Elena Ferrari, and Vijay Atluri. The specification and enforcement of authorization constraints in workflow management systems. *ACM Transactions on Information and System Security*, 2(1):65–104, 1999.

[BGLZ03] Nadia Busi, Roberto Gorrieri, Roberto Lucchi, and Gianluigi Zavattaro. SecSpaces: a data-driven coordination model for environments open to untrusted agents. *Electronic Notes in Theoretical Computer Science*, 68(3):310–327, 2003. Foclasa 2002, Foundations of Coordination Languages and Software Architectures (Satellite Workshop of CONCUR 2002).

[BGSS01] Gregory T. Byrd, Fengmin Gong, Chandramouli Sargor, and Timothy J. Smith. Yalta: A secure collaborative space for dynamic coalitions. In *Proceedings of the 2001 IEEE Workshop on Information Assurance and Security*, pages 30–37. IEEE, 2001.

[BHLR12] Achim D. Brucker, Isabelle Hang, Gero Lückemeyer, and Raj Ruparel. SecureBPMN: Modeling and enforcing access control requirements in business processes. In *Proceedings of the 17th ACM symposium on Access Control Models and Technologies (SACMAT '12)*, pages 123–126. ACM, 2012.

[Bin13] Johann Binder. Introducing the XVSM Micro-Room Framework — creating a privacy preserving peer-to-peer online social network in a declarative way. Master's thesis, TU Wien, 2013.

[Bit15] Lukas Bitter. Design and implementation of a security model for the PeerSpace.NET. Master's thesis, TU Wien, 2015.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture — Volume 1: A System of Patterns*. Wiley Publishing, 1996.

[BMY02] Jean Bacon, Ken Moody, and Walt Yao. A model of OASIS role-based access control and its support for active security. *ACM Transactions on Information and System Security*, 5(4):492–540, 2002.

[BN84] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, 1984.

[BOV99] Ciarán Bryce, Manuel Oriol, and Jan Vitek. A coordination model for agents based on secure spaces. In *COORDINATION 1999: Coordination Languages and Models*, volume 1594 of *LNCS*, pages 4–20. Springer, 1999.

[Bow00] F. D. J. Bowden. A brief survey and synthesis of the roles of time in Petri nets. *Mathematical and Computer Modelling*, 31(10–12):55–68, 2000.

[BP08] Antonio Brogi and Razvan Popescu. Workflow semantics of peer and service behaviour. In *2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE '08)*, pages 143–150. IEEE, 2008.

[BPG+08] Antonio Brogi, Răzvan Popescu, Francisco Gutiérrez, Pablo López, and Ernesto Pimentel. A service-oriented model for embedded peer-to-peer systems. *Electronic Notes in Theoretical Computer Science*, 194(4):5–22, 2008. Proceedings of the 6th International Workshop on the Foundations of Coordination Languages and Software Architectures (FOCLASA 2007).

[BRV02] David Basin, Frank Rittinger, and Luca Viganó. A formal analysis of the CORBA Security Service. In *ZB 2002: Formal Specification and Development in Z and B*, volume 2272 of *LNCS*, pages 330–349. Springer, 2002.

[BZP05] Kevin Borders, Xin Zhao, and Atul Prakash. CPOL: high-performance policy evaluation. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS '05)*, pages 147–157. ACM, 2005.

[Cab14] Albert Caballero. Information security essentials for IT managers: Protecting mission-critical systems. In John R. Vacca, editor, *Managing Information Security*, pages 1–45. Syngress, second edition, 2014.

[Car00] Germano Caronni. Walking the web of trust. In *Proceedings IEEE 9th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE 2000)*, pages 153–158, 2000.

[CCF+00] Fabio Casati, Silvana Castano, Mariagrazia Fugini, Isabelle Mirbel, and Barbara Pernici. Using patterns to design rules in workflows. *IEEE Transactions on Software Engineering*, 26(8):760–785, 2000.

[CCF01] Fabio Casati, Silvana Castano, and MariaGrazia Fugini. Managing workflow authorization constraints through active database technology. *Information Systems Frontiers*, 3(3):319–338, 2001.

[CCG+06] Dario Cerizza, Davide Cerri, Alessandro Ghioni, Geri Joskowicz, Jacek Kopecky, Daniel Martin, Henar Muñoz, Lyndon Nixon, Noelia Pérez, Thorsten Scheibler, and Daniel Wutke. Security and trust requirement analysis and state-of-the-art. TripCom, EU FP6 project, Deliverable D5.1, 2006.

[CCK+09] Davide Cerri, Francesco Corcoglioniti, Jacek Kopecký, Michael Lafite, Kia Teymourian, and Germán Toro del Valle. Final prototype. TripCom, EU FP6 project, Deliverable D5.4, 2009.

[CCM+08] Davide Cerri, Francesco Corcoglioniti, Hans Moritsch, Jacek Kopecký, and Christian Schreiber. Early prototype of the security and trust infrastructure. TripCom, EU FP6 project, Deliverable D5.3, 2008.

[CDJ+13] Stefan Craß, Tobias Dönz, Gerson Joskowicz, eva Kühn, and Alexander Marek. Securing a space-based service architecture with coordination-driven access control. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)*, 4(1):76–97, 2013.

[CDJK12] Stefan Craß, Tobias Dönz, Gerson Joskowicz, and eva Kühn. A coordination-driven authorization framework for space containers. In *Proceedings of the 7th International Conference on Availability, Reliability and Security (ARES'12)*, pages 133–142. IEEE, 2012.

[CDK+02] Francisco Curbera, Matthew Duftler, Rania Khalaf, William Nagy, Nirmal Mukhi, and Sanjiva Weerawarana. Unraveling the Web services web: an introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing*, 6(2):86–93, 2002.

[CDRV06] Michele Cabano, Enrico Denti, Alessandro Ricci, and Mirko Viroli. Designing a BPEL orchestration engine based on ReSpecT tuple centres. *Electronic Notes in Theoretical Computer Science*, 154(1):139–158, 2006. Proceedings of the 4th International Workshop on the Foundations of Coordination Languages and Software Architectures (FOCLASA 2005).

[Cej19] Stephan Cejka. Enabling scalable collaboration by introducing platform-independent communication for the Peer Model. Master's thesis, TU Wien, 2019.

[CF99] Silvana Castano and Maria Grazia Fugini. Rules and patterns for security in workflow systems. In *Database Security XII: Status and Prospects*, volume 14 of *IFIP AICT*, pages 59–74. Springer, 1999.

[CG90] Nicholas Carriero and David Gelernter. *How to Write Parallel Programs: A First Course*. MIT Press, 1990.

[Cha04] David A. Chappell. *Enterprise Service Bus*. O'Reilly Media, 2004.

[CHKS14] Stefan Craß, Jürgen Hirsch, Eva Kühn, and Vesna Sesum-Cavic. Modeling a flexible replication framework for space-based computing. In *ICSOFT 2013: Software Technologies*, volume 457 of *CCIS*, pages 256–272. Springer, 2014.

[CJK15] Stefan Craß, Gerson Joskowicz, and Eva Kühn. A decentralized access control model for dynamic collaboration of autonomous peers. In *SecureComm 2015: Security and Privacy in Communication Networks*, volume 164 of *LNICST*, pages 519–537. Springer, 2015.

[CK12] Stefan Craß and eva Kühn. A coordination-based access control model for space-based computing. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC '12)*, pages 1560–1562. ACM, 2012.

[CKBP14] Stefan Craß, eva Kühn, Sandford Bessler, and Thomas Paulin. A generic load balancing framework for cooperative ITS applications. In *2014 International Conference on Connected Vehicles and Expo (ICCVE)*, pages 385–390. IEEE, 2014.

[CKS09] Stefan Craß, eva Kühn, and Gernot Salzer. Algebraic foundation of a data model for an extensible space-based collaboration protocol. In *Proceedings of the 13th International Database Engineering & Applications Symposium (IDEAS '09)*, pages 301–306. ACM, 2009.

[CKSW17] Stefan Craß, eva Kühn, Vesna Sesum-Cavic, and Harald Watzke. An open event-driven architecture for reactive programming and lifecycle management in space-based middleware. In *43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 189–193. IEEE, 2017.

[CLZ00] Giacomo Cabri, Letizia Leonardi, and Franco Zambonelli. MARS: A programmable coordination architecture for mobile agents. *IEEE Internet Computing*, 4(4):26–35, 2000.

[CMMP06] Paolo Costa, Luca Mottola, Amy L. Murphy, and Gian Pietro Picco. TeenyLIME: transiently shared tuple space middleware for wireless sensor networks. In *Proceedings of the international workshop on Middleware for sensor networks (MidSens '06)*, pages 43–48. ACM, 2006.

[CO03] David W. Chadwick and Alexander Otenko. The PERMIS X.509 role based privilege management infrastructure. *Future Generation Computer Systems*, 19(2):277–289, 2003.

[COZ99] Marco Cremonini, Andrea Omicini, and Franco Zambonelli. Multi-agent systems on the Internet: Extending the scope of coordination towards security and topology. In *MAAMAW 1999: Multi-Agent System Engineering*, volume 1647 of *LNCS*, pages 77–88. Springer, 1999.

[COZ00] Marco Cremonini, Andrea Omicini, and Franco Zambonelli. Coordination and Access Control in Open Distributed Agent Systems: The TuCSoN Approach. In *Proceedings of the 4th International Conference on Coordination Languages and Models (COORDINATION'00)*, volume 1906 of *LNCS*, pages 99–114. Springer, 2000.

[Cra10] Stefan Craß. A formal model of the Extensible Virtual Shared Memory (XVSM) and its implementation in Haskell — design and specification. Master's thesis, TU Wien, 2010.

[Crn01] Ivica Crnkovic. Component-based software engineering — new challenges in software development. *Software Focus*, 2(4):127–133, 2001.

[CSMB05] Bruno Crispo, Swaminathan Sivasubramanian, Pietro Mazzoleni, and Elisa Bertino. P-Hera: Scalable fine-grained access control for P2P infrastructures. In *Proceedings of the 11th International Conference on Parallel and Distributed Systems (ICPADS '05)*, pages 585–591. IEEE, 2005.

[CZO+08] David Chadwick, Gansen Zhao, Sassa Otenko, Romain Laborde, Linying Su, and Tuan Anh Nguyen. PERMIS: A modular authorization infrastructure. *Concurrency and Computation: Practice and Experience*, 20(11):1341–1357, 2008.

[Dij82] Edsger W. Dijkstra. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*, Texts and Monographs in Computer Science, pages 60–66. Springer, 1982.

[Dön11] Tobias Dönz. Design and implementation of the next generation XVSM framework — runtime, protocol and API. Master's thesis, TU Wien, 2011.

[DR08] T. Dierks and E. Rescorla. The transport layer security (TLS) protocol version 1.2. RFC 5246 (Standards Track), 2008. Available at: `https://tools.ietf.org/html/rfc5246`.

[DRV03] Dulce Domingos, António Rito-Silva, and Pedro Veiga. Authorization and access control in adaptive workflows. In *Computer Security — ESORICS 2003*, volume 2808 of *LNCS*, pages 23–38. Springer, 2003.

[DWK01] Dwight Deugo, Michael Weiss, and Elizabeth Kendall. Reusable patterns for agent coordination. In A. Omicini, F. Zambonelli, M. Klusch, and R. Tolksdorf, editors, *Coordination of Internet Agents: Models, Technologies, and Applications*, pages 347–368. Springer, 2001.

[EE14] Asmaa Elkandoussi and Hanan Elbakkali. On access control requirements for inter-organizational workflow. In *Proceedings of the 4th Edition of National Security Days (JNS4)*, pages 1–6. IEEE, 2014.

[EFB01] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming: Introduction. *Communications of the ACM*, 44(10):29–32, 2001.

[EFL+99] C. Elisson, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI certificate theory. RFC 2693, 1999. Available at: `https://tools.ietf.org/html/rfc2693`.

[FAH99] Eric Freeman, Ken Arnold, and Susanne Hupfer. *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley, 1999.

[Fen04] Dieter Fensel. Triple-Space Computing: Semantic web services based on persistent publication of information. In *INTELLCOMM 2004: Intelligence in Communication Systems*, volume 3283 of *LNCS*, pages 43–53. Springer, 2004.

[FHH+99] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. HTTP authentication: Basic and digest access authentication. RFC 2617 (Standards Track), 1999. Available at: `https://tools.ietf.org/html/rfc2617`.

[FHT10] T. Farrell, R. Housley, and S. Turner. An internet attribute certificate profile for authorization. RFC 5755 (Standards Track), 2010. Available at: https://tools.ietf.org/html/rfc5755.

[FLZ06] Riccardo Focardi, Roberto Lucchi, and Gianluigi Zavattaro. Secure shared data-space coordination languages: A process algebraic survey. *Science of Computer Programming*, 63(1):3–15, 2006.

[FMDV07] Tore Fjellheim, Stephen Milliner, Marlon Dumas, and Julien Vayssière. A process-based methodology for designing event-based mobile composite applications. *Data & Knowledge Engineering*, 61(1):6–22, 2007.

[Fon15] Anders Fongen. Data-centric authorization and integrity control in a Linda tuplespace. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC '15)*, pages 1827–1833. ACM, 2015.

[GBK+15] Nikolaos Georgantas, Georgios Bouloukakis, Ajay Kattepur, Mael Besson, Frederic Motte, Fabio Martelli, and Glauco Bigini. CHOReVOLUTION service bus, security and cloud — First outcomes. CHOReVOLUTION, H2020 ICT9 project, Deliverable D3.1, 2015. Available at: http://www.chorevolution.eu/bin/view/Share_Deliverables/WebHome [Accessed 2020-04-09].

[GC92] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, 1992.

[GCC+07] Alessandro Ghioni, Davide Cerri, Francesco Corcoglioniti, Jacek Kopecký, Gerson Joskowicz, Lyndon Nixon, Dario Cerizza, and Noelia Pérez Crespo. Definition of security and trust support model for the reference architecture. TripCom, EU FP6 project, Deliverable D5.2, 2007.

[Gel85] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[GGKL89] Morrie Gasser, Andy Goldstein, Charlie Kaufman, and Butler Lampson. The Digital Distributed System Security Architecture. In *Proceedings of the 1989 National Computer Security Conference*. NIST, 1989.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Gig17] GigaSpaces Technologies. Solutions, Patterns & Best Practices, 2017. Available at: https://docs.gigaspaces.com/sbp/ [Accessed 2020-04-09].

[Gig19a] GigaSpaces Technologies. XAP 14.0 Administration — Security, 2019. Available at: https://docs.gigaspaces.com/xap/14.0/security/ [Accessed 2020-04-09].

[Gig19b] GigaSpaces Technologies. XAP 14.0 Introduction, 2019. Available at: https://docs.gigaspaces.com/xap/14.0/ [Accessed 2020-04-09].

[GLZ06] Roberto Gorrieri, Roberto Lucchi, and Gianluigi Zavattaro. Supporting secure coordination in SecSpaces. *Fundamenta Informaticae*, 73(4):479–506, 2006.

[GM90] Morrie Gasser and Ellen McDermott. An architecture for practical delegation in a distributed system. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 20–30. IEEE, 1990.

[GP03] Daniel Gorla and Rosario Pugliese. Resource access and mobility control with dynamic privileges acquisition. In *ICALP 2003: Automata, Languages and Programming*, volume 2719 of *LNCS*, pages 119–132. Springer, 2003.

[GP04] Daniele Gorla and Rosario Pugliese. Enforcing security policies via types. In *Security in Pervasive Computing*, volume 2802 of *LNCS*, pages 86–100. Springer, 2004.

[GPS99] Paul Grefen, Barbara Pernici, and Gabriel Sánchez, editors. *Database Support for Workflow Management: The WIDE Project*, volume 491 of *The Springer International Series in Engineering and Computer Science*. Springer, 1999.

[Gra78] J.N. Gray. Notes on data base operating systems. In R. Bayer, R.M. Graham, and G. Seegmüller, editors, *Operating Systems*, volume 60 of *LNCS*, pages 393–481. Springer, 1978.

[HA99] Wei-Kuang Huang and Vijayalakshmi Atluri. SecureFlow: a secure web-enabled workflow management system. In *Proceedings of the 4th ACM Workshop on Role-Based Access Control (RBAC '99)*, pages 83–94. ACM, 1999.

[Ham15] Thomas Hamböck. Towards a toolchain for asynchronous embedded programming based on the Peer-Model. Master's thesis, TU Wien, 2015.

[Har10] Shon Harris. *CISSP All-in-One Exam Guide*. McGraw-Hill Osborne Media, 5th edition, 2010.

[Har12] D. Hardt. The OAuth 2.0 authorization framework. RFC 6749 (Standards Track), 2012. Available at: https://tools.ietf.org/html/rfc6749.

[HBAN06] Michael Hafner, Ruth Breu, Berthold Agreiter, and Andrea Nowak. SECTET: an extensible framework for the realization of secure inter-organizational workflows. *Internet Research*, 16(5):491–506, 2006.

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence (IJCAI'73)*, pages 235–245, 1973.

[HBS+13] Mark Hapner, Rich Burridge, Rahul Sharma, Joseph Fialli, Kate Stout, and Nigel Deakin. Java message service (version 2.0). Oracle Specification, 2013. Available at: http://download.oracle.com/otndocs/jcp/jms-2_0-fr-eval-spec/ [Accessed 2020-04-09].

[HCY99] Sandra C. Hayden, Christian Carrick, and Qiang Yang. A catalog of agent coordination patterns. In *Proceedings of the third annual conference on Autonomous Agents (AGENTS '99)*, pages 412–413. ACM, 1999.

[HFS07] Vincent C. Hu, David F. Ferraiolo, and Karen Scarfone. Access control policy combinations for the grid using the policy machine. In *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07)*, pages 225–232. IEEE, 2007.

[HGS+11] Waldemar Hummer, Patrick Gaubatz, Mark Strembeck, Uwe Zdun, and Schahram Dustdar. An integrated approach for identity and access management in a SOA context. In *Proceedings of the 16th ACM symposium on Access control models and technologies (SACMAT '11)*, pages 21–30. ACM, 2011.

[HK03] Patrick C. K. Hung and Kamalakar Karlapalem. A secure workflow model. In *Proceedings of the Australasian Information Security Workshop Conference on ACSW Frontiers 2003*, volume 21 of *Conferences in Research and Practice in Information Technology*, pages 33–41. Australian Computer Society, 2003.

[HR03] Radu Handorean and Gruia-Catalin Roman. Secure sharing of tuple spaces in ad hoc settings. *Electronic Notes in Theoretical Computer Science*, 85(3):122–141, 2003. SecCo'03, First International Workshop on Security Issues in Coordination Models, Languages, and Systems (Satellite Event for ICALP 2003).

[HW04] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2004.

[INC04] INCITS. American National Standard for Information Technology — Role Based Access Control. ANSI INCITS 359-2004, 2004.

[ISO96] ISO/IEC. Information technology — Syntactic metalanguage — Extended BNF. International Standard ISO/IEC 14977:1996, 1996. Available at: https://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996(E).zip.

[ITU12] ITU-T. Information technology — Open Systems Interconnection — The Directory: Public-key and attribute certificate frameworks. International Standard ISO/IEC 9594-8, Recommendation ITU-T X.509 (10/2012), 2012.

Available at: https://www.itu.int/rec/dologin_pub.asp?lang=e&id=T-REC-X.509-201210-S!!PDF-E&type=items.

[JFG+11] Michel Embe Jiague, Marc Frappier, Frédéric Gervais, Régine Laleau, and Richard St-Denis. From ASTD access control policies to WS-BPEL processes deployed in a SOA environment. In *Web Information Systems Engineering — WISE 2010 Workshops*, volume 6724 of *LNCS*, pages 126–141. Springer, 2011.

[JPR05] Christine Julien, Jamie Payton, and Gruia-Catalin Roman. Adaptive access control in coordination-based mobile agent systems. In *SELMAS 2004: Software Engineering for Multi-Agent Systems III*, volume 3390 of *LNCS*, pages 254–271. Springer, 2005.

[JR06] Christine Julien and Gruia-Catalin Roman. EgoSpaces: Facilitating rapid development of context-aware mobile applications. *IEEE Transactions on Software Engineering*, 32(5):281–298, 2006.

[Kan15] Daniel Dimchev Kanev. Decentralized unstructured flat P2P network with streaming content delivery method and user collaboration. Master's thesis, TU Wien, 2015.

[Kar98] Günther Karjoth. Authorization in CORBA security. In *Computer Security — ESORICS 98*, volume 1485 of *LNCS*, pages 143–158. Springer, 1998.

[KC18] eva Kühn and Stefan Craß. Coordination pattern-based approach for auto-scaling in multi-clouds. In *32nd International Conference on Advanced Information Networking and Applications Workshops (WAINA '18), Cloud Computing Project and Initiatives (CCPI)*, pages 368–373. IEEE, 2018.

[KCBŠ19] Eva Kühn, Stefan Craß, Johann Binder, and Vesna Šešum-Čavić. XVSM micro-room process modeler. *International Journal of Cooperative Information Systems*, 28(2), 2019.

[KCH14] eva Kühn, Stefan Craß, and Thomas Hamböck. Approaching coordination in distributed embedded applications with the Peer Model DSL. In *40th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 64–68. IEEE, 2014.

[KCJ98] Lars M. Kristensen, Søren Christensen, and Kurt Jensen. The practitioner's guide to coloured Petri nets. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(2):98–132, 1998.

[KCJ+13] eva Kühn, Stefan Craß, Gerson Joskowicz, Alexander Marek, and Thomas Scheller. Peer-based programming model for coordination patterns. In *COORDINATION 2013: Coordination Models and Languages*, volume 7890 of *LNCS*, pages 121–135. Springer, 2013.

[KCJN14] eva Kühn, Stefan Craß, Gerson Joskowicz, and Martin Novak. Flexible modeling of policy-driven upstream notification strategies. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC '14)*, pages 1352–1354. ACM, 2014.

[KCS15] eva Kühn, Stefan Craß, and Gerald Schermann. Extending a peer-based coordination model with composable design patterns. In *23rd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 53–61. IEEE, 2015.

[KCW10] D. Richard Kuhn, Edward J. Coyne, and Timothy R. Weil. Adding attributes to role-based access control. *Computer*, 43(6):79–81, 2010.

[KFS+99] Myong H. Kang, Judith N. Froscher, Amit P. Sheth, Krys J. Kochut, and John A. Miller. A multilevel secure workflow management system. In *CAiSE 1999: Advanced Information Systems Engineering*, volume 1626 of *LNCS*, pages 271–285. Springer, 1999.

[KM03] Hristo Koshutanski and Fabio Massacci. An access control framework for business processes for web services. In *Proceedings of the 2003 ACM workshop on XML security (XMLSEC '03)*, pages 15–24. ACM, 2003.

[KMKS09] eva Kühn, Richard Mordinyi, László Keszthelyi, and Christian Schreiber. Introducing the concept of customizable structured spaces for agent coordination in the production automation domain. In *Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS '09)*, volume 1, pages 625–632. IFAAMAS, 2009.

[KMS08] Eva Kühn, Richard Mordinyi, and Christian Schreiber. An extensible space-based coordination approach for modeling complex patterns in large systems. In *ISoLA 2008: Leveraging Applications of Formal Methods, Verification and Validation*, volume 17 of *CCIS*, pages 634–648. Springer, 2008.

[KPF01] Myong H. Kang, Joon S. Park, and Judith N. Froscher. Access control mechanisms for inter-organizational workflow. In *Proceedings of the 6th ACM Symposium on Access Control Models and Technologies (SACMAT '01)*, pages 66–74. ACM, 2001.

[KRE18] eva Kühn, Sophie Therese Radschek, and Nahla Elaraby. Distributed coordination runtime assertions for the Peer Model. In *COORDINATION 2018: Coordination Models and Languages*, volume 10852 of *LNCS*, pages 200–219. Springer, 2018.

[KRJ05] eva Kühn, Johannes Riemer, and Gerson Joskowicz. XVSM (eXtensible Virtual Shared Memory) architecture and application. Technical report, Institute of Computer Languages, TU Wien, 2005.

[KS02] Savith Kandala and Ravi Sandhu. Secure role-based workflow models. In *Database and Application Security XV*, volume 87 of *IFIP AICT*, pages 45–58. Springer, 2002.

[Küh12] eva Kühn. Peer Model: Agile middleware and programming model for the coordination of parallel and distributed flows. Technical report, Institute of Computer Languages, TU Wien, 2012.

[Küh16] Eva Kühn. Reusable coordination components: Reliable development of cooperative information systems. *International Journal of Cooperative Information Systems*, 25(4), 2016.

[Küh17] Eva Kühn. Flexible transactional coordination in the Peer Model. In *FSEN 2017: Fundamentals of Software Engineering*, volume 10522 of *LNCS*, pages 116–131. Springer, 2017.

[LABW92] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, 1992.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[Let18] Matthias Lettmayer. A public resource computing application based on the Secure Peer Model. Master's thesis, TU Wien, 2018.

[LKJZ00] S. Li, A. Kittel, D. Jia, and G. Zhuang. Security considerations for workflow systems. In *2000 IEEE/IFIP Network Operations and Management Symposium (NOMS 2000)*, pages 655–668. IEEE, 2000.

[LR14] Maria Leitner and Stefanie Rinderle-Ma. A systematic review on security in process-aware information systems — Constitution, challenges, and future directions. *Information and Software Technology*, 56(3):273–293, 2014.

[LRM11] Maria Leitner, Stefanie Rinderle-Ma, and Juergen Mangler. AW-RBAC: access control in adaptive workflow systems. In *2011 6th International Conference on Availability, Reliability and Security (ARES)*, pages 27–34. IEEE, 2011.

[LZ04] Roberto Lucchi and Gianluigi Zavattaro. WSSecSpaces: a secure data-driven coordination service for web services applications. In *Proceedings of the 2004 ACM Symposium on Applied computing*, pages 487–491. ACM, 2004.

[LZS09] Yahui Lu, Li Zhang, and Jiaguang Sun. Task-activity based access control for process collaboration environments. *Computers in Industry*, 60(6):403–415, 2009.

[MCC+04] R. L. Morgan, Scott Cantor, Steven Carmody, Walter Hoehn, and Ken Klingenstein. Federated security: The Shibboleth approach. *EDUCAUSE Quarterly*, 27(4):12–17, 2004.

[MCSB08] Pietro Mazzoleni, Bruno Crispo, Swaminathan Sivasubramanian, and Elisa Bertino. XACML policy integration algorithms. *ACM Transactions on Information and System Security*, 11(1):4:1–4:29, 2008.

[MK11] Richard Mordinyi and Eva Kühn. Coordination mechanisms in complex software systems. In Nik Bessis and Fatos Xhafa, editors, *Next Generation Data Technologies for Collective Computational Intelligence*, volume 352 of *SCI*, pages 3–30. Springer, 2011.

[ML95] Naftaly H. Minsky and Jerrold Leichter. Law-Governed Linda as a coordination model. In *Object-Based Models and Languages for Concurrent Systems — ECOOP '94 Workshop on Models and Languages for Coordination of Parallelism and Distribution*, volume 924 of *LNCS*, pages 125–146. Springer, 1995.

[MMD+14] Mukhtiar Memon, Gordhan D. Menghwar, Mansoor H. Depar, Akhtar A. Jalbani, and Waqar M. Mashwani. Security modeling for service-oriented systems using security pattern refinement approach. *Software & Systems Modeling*, 13(2):549–572, 2014.

[MMU00] Naftaly H. Minsky, Yaron M. Minsky, and Victoria Ungureanu. Making tuple spaces safe for heterogeneous distributed systems. In *Proceedings of the 2000 ACM Symposium on Applied Computing (SAC '00)*, volume 1, pages 218–226. ACM, 2000.

[MSB11a] Jutta Mülle, Silvia von Stackelberg, and Klemens Böhm. Modelling and transforming security constraints in privacy-aware business processes. In *2011 IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*. IEEE, 2011.

[MSB11b] Jutta Mülle, Silvia von Stackelberg, and Klemens Böhm. A security language for BPMN process models. Karlsruhe Reports in Informatics, Karlsruhe Institute of Technology (KIT), 2011.

[MWL08] Daniel Martin, Daniel Wutke, and Frank Leymann. A novel approach to decentralized workflow enactment. In *2008 12th International IEEE Enterprise Distributed Object Computing Conference (EDOC '08)*, pages 127–136. IEEE, 2008.

[MZ06] A. Melnikov and K. Zeilenga. Simple authentication and security layer (SASL). RFC 4422 (Standards Track), 2006. Available at: `https://tools.ietf.org/html/rfc4422`.

[Nat01] National Institute of Standards and Technology (NIST). Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197, 2001. Available at: https://www.nist.gov/publications/advanced-encryption-standard-aes.

[NBB+60] Peter Naur, J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H Wegstein, A. van Wijngaarden, and M. Woodger. Report on the algorithmic language ALGOL 60. *Communications of the ACM*, 3(5):299–314, 1960.

[NFP98] Rocco De Nicola, Gian Luigi Ferrari, and Rosario Pugliese. KLAIM: A kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.

[NFP99] Rocco De Nicola, GianLuigi Ferrari, and Rosario Pugliese. Types as specifications of access policies. In Jan Vitek and Christian D. Jensen, editors, *Secure Internet Programming*, volume 1603 of *LNCS*, pages 117–146. Springer, 1999.

[NG11] Hema Andal Jayaprakash Narayanan and Mehmet Hadi Güneş. Ensuring access control in cloud provisioned healthcare systems. In *2011 IEEE Consumer Communications and Networking Conference (CCNC)*, pages 247–251. IEEE, 2011.

[NT94] B. Clifford Neuman and Theodore Ts'o. Kerberos: an authentication service for computer networks. *IEEE Communications Magazine*, 32(9):33–38, 1994.

[NYHR05] C. Neuman, T. Yu, S. Hartman, and K. Raeburn. The Kerberos network authentication service (v5). RFC 4120 (Standards Track), 2005. Available at: https://tools.ietf.org/html/rfc4120.

[OA11] OVE and Austrian Standards Institute. Railway Applications — Communication, signalling and processing systems — Safety-related communication in transmission systems. ÖVE/ÖNORM EN 50159: 2011 05 01, 2011.

[OAS05] OASIS Security Services TC. Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2.0. OASIS Standard, 2005. Available at: http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf.

[OAS07] OASIS Web Services Business Process Execution Language (WSBPEL) TC. Web Services Business Process Execution Language Version 2.0. OASIS Standard, 2007. Available at: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

[OAS08] OASIS Security Services TC. Security Assertion Markup Language (SAML) V2.0 Technical Overview. Committee Draft 02, 2008. Available at: http://docs.oasis-open.org/security/saml/Post2.0/sstc-saml-tech-overview-2.0-cd-02.html.

[OAS09a] OASIS Security Services TC. SAML V2.0 Condition for Delegation Restriction Version 1.0. Committee Specification 01, 2009. Available at: http://docs.oasis-open.org/security/saml/Post2.0/sstc-saml-delegation-cs-01.pdf.

[OAS09b] OASIS Web Services Federation (WSFED) TC. Web Services Federation Language (WS-Federation) Version 1.2. OASIS Standard, 2009. Available at: http://docs.oasis-open.org/wsfed/federation/v1.2/os/ws-federation-1.2-spec-os.pdf.

[OAS12a] OASIS Web Services Secure Exchange (WS-SX) TC. WS-Trust 1.4. OASIS Standard, 2012. Available at: http://docs.oasis-open.org/ws-sx/ws-trust/v1.4/errata01/os/ws-trust-1.4-errata01-os-complete.pdf.

[OAS12b] OASIS Web Services Security Maintenance (WSS-M) TC. Web Services Security: SOAP Message Security Version 1.1.1. OASIS Standard, 2012. Available at: http://docs.oasis-open.org/wss-m/wss/v1.1.1/os/wss-SOAPMessageSecurity-v1.1.1-os.pdf.

[OAS13] OASIS eXtensible Access Control Markup Language (XACML) TC. eXtensible Access Control Markup Language (XACML) Version 3.0. OASIS Standard, 2013. Available at: http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.pdf.

[OAS14a] OASIS eXtensible Access Control Markup Language (XACML) TC. XACML v3.0 Administration and Delegation Profile Version 1.0. Committee Specification Draft 04, 2014. Available at: http://docs.oasis-open.org/xacml/3.0/administration/v1.0/csd04/xacml-3.0-administration-v1.0-csd04.pdf.

[OAS14b] OASIS eXtensible Access Control Markup Language (XACML) TC. XACML v3.0 Core and Hierarchical Role Based Access Control (RBAC) Profile Version 1.0. Committee Specification 02, 2014. Available at: http://docs.oasis-open.org/xacml/3.0/rbac/v1.0/cs02/xacml-3.0-rbac-v1.0-cs02.pdf.

[Obj02] Object Management Group. Security Service Specification, Version 1.8, 2002. Available at: http://www.omg.org/spec/SEC/1.8/.

[Obj04] Object Management Group. Common Object Request Broker Architecture: Core Specification, Version 3.0.3, 2004. Available at: http://www.omg.org/spec/CORBA/3.0.3/.

[Obj11] Object Management Group. Business Process Model and Notation (BPMN), Version 2.0, 2011. Available at: http://www.omg.org/spec/BPMN/2.0/.

[Obj15] Object Management Group. OMG Unified Modeling Language (OMG UML), Version 2.5, 2015. Available at: http://www.omg.org/spec/UML/2.5/.

[OH05] Manuel Oriol and Michael Hicks. Tagged sets: A secure and transparent coordination medium. In *COORDINATION 2005: Coordination Models and Languages*, volume 3454 of *LNCS*, pages 252–267. Springer, 2005.

[OP03] Sejong Oh and Seog Park. Task–role-based access control model. *Information Systems*, 28(6):533–562, 2003.

[OPA07] Lukasz Opyrchal, Atul Prakash, and Amit Agrawal. Supporting privacy policies in a publish-subscribe substrate for pervasive environments. *Journal of Networks*, 2(1):17–26, 2007.

[Ope07] OpenID. OpenID Authentication 2.0. OpenID Specification, 2007. Available at: http://openid.net/specs/openid-authentication-2_0.html [Accessed 2020-04-09].

[OR03] Andrea Omicini and Alessandro Ricci. Reasoning about organisation: Shaping the infrastructure. *AI*IA Notizie*, 16(2):7–16, 2003.

[Ora19] Oracle. Java Authentication and Authorization Service (JAAS) Reference Guide. Java SE JDK 12 Documentation — Security Guide, 2019. Available at: https://docs.oracle.com/en/java/javase/12/security/java-authentication-and-authorization-service-jaas-reference-guide.html [Accessed 2020-04-09].

[ORV05] Andrea Omicini, Alessandro Ricci, and Mirko Viroli. RBAC for organisation and security in an agent coordination infrastructure. *Electronic Notes in Theoretical Computer Science*, 128(5):65–85, 2005. Proceedings of the 2nd International Workshop on Security Issues in Coordination Models, Languages, and Systems (SecCo 2004).

[PA98] George A. Papadopoulos and Farhad Arbab. Coordination models and languages. In Marvin V. Zelkowitz, editor, *Advances in Computers*, volume 46, pages 329–400. Elsevier, 1998.

[PB02] Peter R. Pietzuch and Jean M. Bacon. Hermes: A distributed event-based middleware architecture. In *Proceedings 22nd International Conference on Distributed Computing Systems Workshops*, pages 611–618. IEEE, 2002.

[PBC08] Federica Paci, Elisa Bertino, and Jason Crampton. An access-control framework for WS-BPEL. *International Journal of Web Services Research (IJWSR)*, 5(3):20–43, 2008.

[PEB06] Lauri I. W. Pesonen, David M. Eyers, and Jean Bacon. A capability-based access control architecture for multi-domain publish/subscribe systems. In *Proceedings International Symposium on Applications and the Internet (SAINT '06)*, pages 222–228. IEEE, 2006.

[Pet66] Carl Adam Petri. Communication with automata. Technical Report RADC-TR-65-377, Volume I, Final Report, Supplement I, Rome Air Development Center, 1966.

[PMR99] Gian Pietro Picco, Amy L. Murphy, and Gruia-Catalin Roman. LIME: Linda meets mobility. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 368–377. ACM, 1999.

[Rau14] Dominik Rauch. PeerSpace.NET — implementing and evaluating the Peer Model with focus on API usability. Master's thesis, TU Wien, 2014.

[RD09] Giovanni Russello and Naranker Dulay. xDUCON: Coordinating usage control policies in distributed domains. In *Proceedings 3rd International Conference on Network and System Security (NSS '09)*, pages 246–253. IEEE, 2009.

[RDD08] Giovanni Russello, Changyu Dong, and Naranker Dulay. A workflow-based access control framework for e-health applications. In *22nd International Conference on Advanced Information Networking and Applications — Workshops (AINAW 2008)*, pages 111–120. IEEE, 2008.

[RMD+06] Johannes Riemer, Francisco Martin-Recuerda, Ying Ding, Martin Murth, Brahmananda Sapkota, Reto Krummenacher, Omair Shafiq, Dieter Fensel, and Eva Kühn. Triple Space Computing: adding semantics to space-based computing. In *The Semantic Web — ASWC 2006*, volume 4185 of *LNCS*, pages 300–306. Springer, 2006.

[ROD02] Alessandro Ricci, Andrea Omicini, and Enrico Denti. Virtual enterprises and workflow management as agent coordination issues. *International Journal of Cooperative Information Systems*, 11(3–4):355–379, 2002.

[SBJ+14] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, and C. Mortimore. OpenID Connect Core 1.0. OpenID Specification, 2014. Available at: `http://openid.net/specs/openid-connect-core-1_0.html` [Accessed 2020-04-09].

[SCFY96] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.

[Sch06] Douglas C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31, 2006.

[Sch13] Thomas Schmid. Dynamic migration of cloud services on the basis of changeable parameters. Master's thesis, TU Wien, 2013.

[Sch17] Jörg Schoba. Mobile Peer Model — a mobile peer-to-peer communication and coordination framework — with focus on scalability and security. Master's thesis, TU Wien, 2017.

[SFH+06] Markus Schumacher, Eduardo Fernandez-Buglioni, Duane Hybertson, Frank Buschmann, and Peter Sommerlad. *Security Patterns: Integrating Security and Systems Engineering*. Wiley, 2006.

[SGP15] Amleto Di Salle, Francesco Gallo, and Alexander Perucci. Towards adapting choreography-based service compositions through enterprise integration patterns. In *Software Engineering and Formal Methods (SEFM 2015 Workshops)*, volume 9509 of *LNCS*, pages 240–252. Springer, 2015.

[SKN07] Elena Simperl, Reto Krummenacher, and Lyndon Nixon. A coordination model for triplespace computing. In *COORDINATION 2007: Coordination Models and Languages*, volume 4467 of *LNCS*, pages 1–18. Springer, 2007.

[SS75] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[TAK03] Anand R. Tripathi, Tanvir Ahmed, and Richa Kumar. Specification of secure distributed collaboration systems. In *The Sixth International Symposium on Autonomous Decentralized Systems (ISADS 2003)*, pages 149–156. IEEE, 2003.

[TAPH05] William Tolone, Gail-Joon Ahn, Tanusree Pai, and Seng-Phil Hong. Access control in collaborative systems. *ACM Computing Surveys*, 37(1):29–41, 2005.

[Til17] Peter Tillian. Mobile Peer Model — a mobile peer-to-peer communication and coordination framework — with focus on mobile design constraints. Master's thesis, TU Wien, 2017.

[Tol98] Robert Tolksdorf. Coordination patterns of mobile information agents. In *Cooperative Information Agents II: Learning, Mobility and Electronic Commerce for Information Discovery on the Internet (CIA 1998)*, volume 1435 of *LNCS*, pages 246–261. Springer, 1998.

[TS98] R. K. Thomas and R. S. Sandhu. Task-based authorization controls (TBAC): a family of models for active and enterprise-oriented authorization management. In *Database Security XI: Status and Prospects*, IFIP AICT, pages 166–181. Springer, 1998.

[TV03] Robin D. Toll and Carlos Varela. Mobility and security in worldwide computing. In *Proceedings of the 9th ECOOP Workshop on Mobile Object Systems*, 2003.

[UFF15] Anton V. Uzunov, Eduardo B. Fernandez, and Katrina Falkner. Security solution frames and security patterns for authorization in distributed, collaborative systems. *Computers & Security*, 55:193–234, 2015.

[UWJ07] Nur Izura Udzir, Alan M. Wood, and Jeremy L. Jacob. Coordination with multicapabilities. *Science of Computer Programming*, 64(2):205–222, 2007.

[VAC08] Claudio Vairo, Michele Albano, and Stefano Chessa. A secure middleware for wireless sensor networks. In *Proceedings of the 5th Annual International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services (MobiQuitous 2008)*. ICST, 2008.

[VBO03] Jan Vitek, Ciarán Bryce, and Manuel Oriol. Coordinating processes with secure spaces. *Science of Computer Programming*, 46(1-2):163–193, 2003.

[VM12] Wattana Viriyasitavat and Andrew Martin. A Survey of Trust in Workflows and Relevant Contexts. *IEEE Communications Magazine*, 14(3):911–940, 2012.

[W3C17] W3C. XML Path Language (XPath) 3.1. W3C Recommendation, 2017. Available at: `https://www.w3.org/TR/xpath-31/`.

[Wal98] Jim Waldo. Remote procedure calls and java remote method invocation. *IEEE Concurrency*, 6(3):5–7, 1998.

[WCB01] Matt Welsh, David Culler, and Eric Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 230–243. ACM, 2001.

[Win11] Markus Winkler. XIDS — an XVSM-based collaborative intrusion detection system. Master's thesis, TU Wien, 2011.

[WKB07] Jacques Wainer, Akhil Kumar, and Paulo Barthelmess. DW-RBAC: A formal security model of delegation and revocation in workflow systems. *Information Systems*, 32(3):365–384, 2007.

[WML08] Daniel Wutke, Daniel Martin, and Frank Leymann. Model and infrastructure for decentralized workflow enactment. In *Proceedings of the 2008 ACM Symposium on Applied Computing (SAC '08)*, pages 90–94. ACM, 2008.

[WMLF98] P. Wyckoff, S. W. McLaughry, T. J. Lehman, and D. A. Ford. T spaces. *IBM Systems Journal*, 37(3):454–474, 1998.

[WPJV03] Bart De Win, Frank Piessens, Wouter Joosen, and Tine Verhanneman. On the importance of the separation-of-concerns principle in secure software engineering. In *ACSA Workshop on the Application of Engineering Principles to System Security Design — Final Report*, 2003.

[Wri11] Alex Wright. Hacking cars. *Communications of the ACM*, 54(11):18–19, 2011.

[WSML02] Shengli Wu, Amit P. Sheth, John A. Miller, and Zongwei Luo. Authorization and access control of application data in workflow systems. *Journal of Intelligent Information Systems*, 18(1):71–94, 2002.

[WV04] Jehan Wickramasuriya and Nalini Venkatasubramanian. Dynamic access control for ubiquitous environments. In *On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE (OTM 2004)*, volume 3291 of *LNCS*, pages 1626–1643. Springer, 2004.

[YKM14] Younis A. Younis, Kashif Kifayat, and Madjid Merabti. An access control model for cloud computing. *Journal of Information Security and Applications*, 19(1):45–60, 2014.

[YT05] Eric Yuan and Jin Tong. Attributed based access control (ABAC) for Web services. In *Proc. 2005 IEEE International Conference on Web Services (ICWS 2005)*. IEEE, 2005.

[ZBH10] Mahdi Zargayouna, Flavien Balbo, and Serge Haddad. Data driven language for agents secure interaction. In *Languages, Methodologies, and Development Tools for Multi-Agent Systems (LADS 2009)*, volume 6039 of *LNCS*. Springer, 2010.

[Zei06] K. Zeilenga. Lightweight directory access protocol (LDAP): Technical specification road map. RFC 4510 (Standards Track), 2006. Available at: `https://tools.ietf.org/html/rfc4510`.

Stefan Craß

Curriculum Vitae

Personal Information:

Date of birth: April 2nd, 1984
Place of birth: Eisenstadt, Austria
Citizenship: Austrian
Address: Gassergasse 32/16, 1050 Wien, Austria
Email: stefan.crass@gmail.com

Employment:

since 2020: Senior Researcher at the Austrian Blockchain Center (ABC Research GmbH)

2010 – 2020: Research Assistant at the Space-Based Computing Group (Prof. Eva Maria Kühn), TU Wien, Institute of Information Systems Engineering (until 2017: Institute of Computer Languages), Research Division "Compilers and Languages"

2007 – 2010: Tutor for the course "Distributed Programming with Space Based Computing Middleware" at TU Wien

Education:

2010 – 2020: Doctoral studies in Computer Science at TU Wien

2007 – 2010: Master studies "Software Engineering & Internet Computing" at TU Wien, graduation with distinction as "Dipl.-Ing." (Thesis: "Formal Model of XVSM and Implementation of a Haskell Prototype")

2003 – 2007: Bachelor studies "Software & Information Engineering" at TU Wien, graduation with distinction as "BSc"

Selected Publications:

- Stefan Craß, eva Kühn, and Gernot Salzer. Algebraic foundation of a data model for an extensible space-based collaboration protocol. *International Database Engineering & Applications Symposium (IDEAS '09)*, pages 301-306. ACM, 2009.
- Stefan Craß and eva Kühn. A coordination-based access control model for space-based computing. *ACM Symposium on Applied Computing (SAC '12)*, pages 1560-1562. ACM, 2012.
- Stefan Craß, Tobias Dönz, Gerson Joskowicz, and eva Kühn. A coordination-driven authorization framework for space containers. *International Conference on Availability, Reliability and Security (ARES '12)*, pages 133-142. IEEE, 2012.

- Stefan Craß, Tobias Dönz, Gerson Joskowicz, eva Kühn, and Alexander Marek. Securing a space-based service architecture with coordination-driven access control. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)*, 4(1):76-97, 2013.
- eva Kühn, Stefan Craß, Gerson Joskowicz, Alexander Marek, and Thomas Scheller. Peer-based programming model for coordination patterns. *COORDINATION 2013: Coordination Models and Languages*, volume 7890 of *LNCS*, pages 121-135. Springer, 2013.
- Stefan Craß, eva Kühn, Sandford Bessler, and Thomas Paulin. A generic load balancing framework for cooperative ITS applications**.** *International Conference on Connected Vehicles & Expo (ICCVE)*, pages 385-390. IEEE, 2014.
- eva Kühn, Stefan Craß, and Thomas Hamböck. Approaching Coordination in Distributed Embedded Applications with the Peer Model DSL**.** *Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 64-68. IEEE, 2014.
- Stefan Craß, Jürgen Hirsch, eva Kühn, and Vesna Šešum-Čavić. Modeling a flexible replication framework for space-based computing**.** *Communications in Computer and Information Science (CCIS), ICSOFT-2013*, Volume 457, pages 256-272. Springer, 2014.
- eva Kühn, Stefan Craß, and Gerald Schermann. Extending a peer-based coordination model with composable design patterns. *Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 53-61. IEEE, 2015.
- Stefan Craß, Gerson Joskowicz, and Eva Kühn. A decentralized access control model for dynamic collaboration of autonomous peers. *SecureComm 2015: Security and Privacy in Communication Networks*, volume 164 of *LNICST*, pages 519-537. Springer, 2015.
- Stefan Craß, eva Kühn, Vesna Šešum-Čavić, and Harald Watzke. An open event-driven architecture for reactive programming and lifecycle management in space-based middleware. *Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 189-193. IEEE, 2017.
- eva Kühn and Stefan Craß. Coordination pattern-based approach for auto-scaling in multi-clouds. *International Conference on Advanced Information Networking and Applications Workshops (WAINA'18), Cloud Computing Project and Initiatives (CCPI)*, pages 368-373. IEEE, 2018.
- Eva Kühn, Stefan Craß, Johann Binder, and Vesna Šešum-Cavic. XVSM micro-room process modeler. *International Journal of Cooperative Information Systems*, 28(2), 2019.