FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Design and Implementation of a Security Model for the PeerSpace.NET

MAGISTERARBEIT

zur Erlangung des akademischen Grades

Magister

im Rahmen des Studiums

Informatikmanagement

eingereicht von

Lukas Bitter
Matrikelnummer 0425266

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. eva Kühn
Mitwirkung: Projektass. Dipl.-Ing. Stefan Craß

Wien, 27.9.2015

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Design and Implementation of a Security Model for the PeerSpace.NET

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Magister

in

Informatikmanagement

by

Lukas Bitter
Registration Number 0425266

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. eva Kühn
Assistance: Projektass. Dipl.-Ing. Stefan Craß

Vienna, 27.9.2015

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Lukas Bitter
Schulzgasse 15/18 in 1210 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.

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

I like to thank my professor eva Kühn and her research assistant Stefan Craß for facilitating the work on this interesting topic, as well as supporting me by providing constructive and inspiring feedback.

Furthermore, I would like to thank my mom, dad, aunt and uncle of whom each supported me in their own way. Last but not least, I am grateful for the support of certain friends for their support and motivation during the process of creating this thesis.

Abstract

The users of space-based applications are decoupled from each other in time and space because they interact with each other by writing and reading/taking objects to/from the space. This property of space-based applications is useful for modern applications.

The Peer Model is currently the highest abstraction of the space-based paradigm and facilitates the creation of reusable coordination patterns with the embedding of decoupled application logic. The Peer Model's implementation in .Net is called PeerSpace.NET. Currently there exists no security mechanism for the PeerSpace.NET, which is important for its practical employment.

This thesis discusses the creation and implementation of a security model for the PeerSpace.NET. Due to the present P2P architecture, where no centralized server exists and no mutual trust can be assumed, several challenges arise for the creation of the security model.

The here presented security model protects the PeerSpace.NET against unauthorized access by means of a fine-grained policy. The access control is based on authenticated security attributes which identify the sender of entries. To facilitate access control for entries which are sent on behalf of other peers, indirect senders are also identified by their security attributes.

Access control decisions, i.e. granting or denying an operation, involves information about the content of sent entries and may depend on environmental context data. Further a peer's security policy can be dynamically changed by the peer owner but the security administration can also be delegated to other users.

In a nutshell, a security model with a dynamic, content- and context-aware access control, which can also involve indirect senders for its security decision, is created and presented throughout this thesis.

Kurzfassung

Space-basierte Computersysteme entkoppeln ihre Nutzer bezüglich Ort und Zeit, da diese nicht direkt miteinander interagieren, sondern indem sie Objekte in den Space schreiben, respektive von diesem lesen oder nehmen. Aufgrund dieser Eigenschaft der Entkoppelung sind moderne Applikationen oft space-basiert.

Das Peer Model ist gegenwärtig die höchste Abstraktion des space-basierten Programmiermusters. Es ermöglicht die Erstellung von wiederverwendbaren Koordinationsvorlagen und davon entkoppelten Serviceaufrufen. Für die .NET-Implementierung des Peer Models (PeerSpace-.NET) gibt es momentan keine Security-Mechanismen, die aber für den praktischen Einsatz wichtig wären.

Diese Arbeit beschäftigt sich mit der Erstellung und Implementierung eines Security Models für den PeerSpace.NET. Dieses Security Model schützt den PeerSpace.NET vor unbefugten Zugriffen mithilfe von präzise definierbaren Regeln. Diese Zugriffsregelung basiert auf authentifizierten Security-Attributen, mit denen die Sender von Entries identifiziert werden. Es ist auch möglich Regeln für den Fall zu definieren, dass Entries im Auftrag eines anderen Users gesendet wurden. Diese Regeln benutzen dann die Identität des Auftraggebers (indirekter Sender) und die des direkten Senders, um Zugriffsrechte zu definieren.

Regeln können auch Eigenschaften von Entries, deren Zugriff sie regeln, in die Definition der Zugriffsrechte miteinbeziehen. Weiteres kann auch der Zustand des PeerSpace.NET in die Entscheidung, ob ein Zugriff für Entries gewährt wird oder nicht, miteinbezogen werden.

Die Zugriffsrechte zu einem Peer werden von dessen Eigentümer verwaltet. Der Eigentümer kann aber auch einen anderer Benutzer mit der Verwaltung der Zugriffsrechte beauftragen.

Zusammengefasst wird in dieser Arbeit ein Security Model für den PeerSpace.NET entwickelt und implementiert, das inhalts- und zustandsabhängige Zugriffsentscheidungen definieren kann und auch die Identität von direkten und indirekten Sendern miteinbezieht.

Contents

List of Figures

List of Tables

List of Listings

CHAPTER 1

Introduction

Distributed computer systems, providing various services and data, play an essential role in the business community, as well as in the technical and the private field, since they enable the cooperation of multiple users possibly resident around the world. Users may join and/or exit a distributed computer system frequently at any time and thus the system may change its size in a highly dynamic way. Further the users may join the computer system with different devices and are naturally not permanently online.

Thus the requirements for modern distributed computer systems have heavily increased compared to former times. Modern distributed computer systems must be capable to integrate heterogeneous devices in a highly dynamic way. Further these systems must be reliable and scalable.

While the client-server architecture was mainly deployed in earlier days, this approach does not meet the requirements of modern systems as a dedicated server depicts a bottleneck and thus scalability is affected. Further one dedicated server constitutes a single point of failure and consequently the reliability of such a system is not high.

In the space-based paradigm, the application is built in terms of autonomous units which interact by reading, taking and writing objects from/to a space. This decouples the participants from each other in time and place, which facilitates an asynchronous communication and avoids the necessity that attendees must know each other's addresses. Furthermore, this paradigm supports the collaboration of heterogeneous units. Thus the space-based paradigm satisfies some of the requirements of modern distributed computer systems.

The Linda Tuple Space [1] is a programming model according to the space-based paradigm, where the units read, take and write so-called tuples from/to a space. Thereby read and take operations are conducted with the aid of template matching, whereby objects are selected from the space by means of templates. A template defines values for specified fields of an object and every object from the space that possesses the same number of fields with the same types and the same values as the template is potentially selected. A defined count of matching objects is read or taken. The original Linda Tuple Space programming model follows the client-server approach, where the server hosts the space and the clients are the autonomously interacting with

each other via the space. Accordingly the original Linda Tuple Space model does not satisfy reliability and scalability due to the client-server architecture.

The decentralized peer-to-peer (P2P) architecture satisfies reliability and scalability, as every entity of the network can be a client and/or a server. There is no bottleneck and thus scalability is given and there is no single point of failure and thus reliability is given.

The concepts of the Linda Tuple Space and the P2P architecture have been merged. The result is the distributed space architecture which combines the advantages of both concepts. Thus distributed space architecture models satisfy the requirements of modern distributed computer systems.

The implementation of the coordination logic in P2P architectures is a very complex task, as no dedicated server is available and the network may change in a highly dynamic way. Further, (distributed) transactions in such dynamic architectures are not trivial and error-prone.

There exist several frameworks that implement the P2P architecture. Such middlewares serve as basis for the implementation of distributed systems and solve recurrent issues like synchronization and transactions and thus the programmer can focus on the actual implementation and may reuse proven mechanisms.

The eXtensible Virtual Shared Memory (XVSM) [2] is a space-based framework developed at the Institute of Computer Languages of the Vienna University of Technology and is an enhancement to the Linda Tuple Space. XVSM structures the space into sub-spaces called containers and provides multiple coordinators which define how entries are written and queried.

The Peer Model [3], which has also been developed at the very same institute, is currently the highest abstraction of the space-based paradigm. It allows to build diverse and even complex data flow patterns, regardless of the application logic, which is realized with arbitrary service methods. This model strictly encapsulates the application and the coordination logic which results in a good maintainability, reusability, extensibility and scalability.

The Peer Model is composed of *Peers*, *Containers*, *Wirings* and *Entries*. A peer possesses one input container (PIC), one output container (POC) and an arbitrary number of wirings, which are the centerpiece of a peer. A wiring states a query for certain entries against a source container and performs arbitrary service methods when the query is fulfilled. After all methods have been processed, the wiring writes a defined subset of entries to stated destination containers. The choice of the containers for the source and the destination are strictly encapsulated from the service methods which facilitates the already mentioned encapsulation of coordination and application logic.

There exists an implementation of the Peer Model in C# for the .NET platform, which is called PeerSpace.NET [4]. Due to the P2P-like architecture of the PeerSpace.NET, several security threats occur, which are *eavesdropping, traffic analysis, spoofing, man-in-the-middle* and *replay attacks* [5, 6]. For the practical employment of the PeerSpace.NET an appropriate mechanism in order to secure the PeerSpace.NET against different forms of attacks is needed. The design and implementation of a security model which protects the PeerSpace.NET forms the goal of this thesis.

There exist diverse access control models, but they are not designed for distributed systems and thus they are not expressive enough for the PeerSpace.NET. Thus, an appropriate security model will be designed and implemented into the already existing PeerSpace.NET. Since the

users do not reside within a trusted domain where authentication and authorization is achieved by a dedicated server and mutual trust cannot be assumed, designing the security model for the PeerSpace.NET is more challenging than for a classical client-server environment. The basic requirements for the security model do not differ from those of other computer systems and are as follows: authentication, authorization, confidentiality, integrity and non-repudiation. As the PeerSpace.NET is a distributed middleware in the style of P2P networks, where no trusted server exists, an appropriate mechanism for authenticating the users must be found and deployed. Unauthorized sending of entries and thus illegal service invocations will be prohibited with the aid of appropriate access control. As the Peer Model allows to forward entries from peer to peer, the security model must be capable to express access control rules that involve the identity of all senders of a forwarded entry. Fine-grained rules are important to meet the principle of least privilege, whereby entities obtain exactly the privileges they need to conduct their task and not more [7]. As the security requirements may change dynamically it is useful to facilitate users the self-administration of their security policy [8]. Further the possibility to delegate the administration of the security policy to certain users is handy, as an expert may conduct this task.

The basis for the security model for the PeerSpace.NET builds the Secure Peer Model introduced in [9] which is a sophisticated security model for the Peer Model. The Secure Peer Model is adapted to the PeerSpace.NET and simplified due to practicability but it is still capable to express fine-grained rules.

The research question is whether the implementation of such a security model for the PeerSpace.NET is feasible and which grade of usability can be achieved for the application of the security model. Further, the performance overhead will be evaluated in order to verify the practicability of the developed security model.

1.1 Methodology

After defining the goal of this thesis, literature researches concerning authentication, authorization and middlewares are conducted. Further, the Peer Model, the PeerSpace.NET and the Secure Peer Model are examined and the requirements for the security model for the PeerSpace-.NET are compiled. With aid of the knowledge gained from the literature research, appropriate concepts and mechanisms are adapted for the PeerSpace.NET and the design is created. After the design has been evaluated concerning the requirements it is implemented in the PeerSpace.NET. Finally, the implementation is evaluated by means of a theoretical inspection of the fulfilled requirements, a use case which demonstrates most of the functionality of the security model, a benchmark test to evaluate the performance and a comparison to the security models of other middlewares.

1.2 Structure of this Thesis

The introduction is followed by chapter 2 which represents the literature review. Chapter 3 gives an insight into the Peer Model, its implementation in .NET and the Secure Peer Space. Further, the used technologies are briefly discussed there. All requirements for the security model for the PeerSpace.NET are listed in chapter 4. Chapter 5 explains the architecture of the security

model as well as the used mechanisms. Chapter 6 presents the implementation of the security model and explains how it interacts with the mechanisms of the PeerSpace.NET. The evaluation is outlined in chapter 7 and contains the use case, a benchmark test and the comparison to the security features of the middlewares from the related work. Chapter 8 presents the future work and suggests possible enhancements of the developed security model. This thesis concludes with chapter 9.

CHAPTER 2

Related Work

This section is thought to give an overview about the topic of this thesis and analyze existing systems and mechanisms concerning authentication and authorization. Further several systems for distributed computing are analyzed concerning their security functionalities, which are subsequently compared. Finally, the Peer Model, the PeerSpace.NET and the Secured Peer Space are examined. With aid of the gathered knowledge sound decisions for the design of the security model can be made.

2.1 Authentication and Authorization Systems and Protocols

In the context of distributed computer systems, authentication is the verification of the identity of the interacting opposite entity [10]. In order to obtain an overview of the state-of-the-art authentication and/or authorization mechanisms, the following systems and protocols are analyzed.

2.1.1 OAuth

OAuth [11] is a token-based standardized open protocol for authorization delegation from web, mobile and desktop applications. A typical scenario for achieving an authorization delegation through OAuth is as follows:

Assume a person wants to give a social network website access rights to his/her email account to find friends by means of known email addresses. From a security perspective it would be dire if the person exposed its email credential to the social network website. OAuth facilitates that the social network gains access to the contacts from the email account without passing the credentials.

In order to discuss the functionality of OAuth some definitions of terms, used in the context with OAuth, are useful. The OAuth provider is a server where the protected resource is located, which is the mail server in this case. The OAuth client is a user, a web service or an application, that wants to gain access to the protected resource. In this case the OAuth client is the social network website and the protected resources are the email addresses. First the client has to be

registered at the provider. It receives through the registration process a client id token and a client secret token. This procedure cannot be automated and must be executed only once.

Let us consider the example wherein the user is logged in at the social network website, which is the OAuth client and wants to permit that site to gain access to the mail server representing the OAuth provider. Therefore the user is sent to the mail server along with the client id token and a client URL for response. At the mail server the locally authenticated user is asked if he/she wants the social network site to gain access to the mail server. When the user agrees, he/she will be redirected back to the client along with a temporary token. The client takes that token, issues a token that contains the temporary token, the client id token and the client secret token and sends it back to the provider. This handshake is to ensure that the user wants that client to be permitted to gain access to the provider. Then the mail server sends back an access token to the social network site, whereby it gains access to the mail server.

2.1.2 OpenID

OpenID is a standardized decentralized protocol for authentication, which facilitates the user to authenticate him/her at a website with the aid of an OpenID. This has the advantage that the user must only administrate one account at the OpenID provider and may use it for the authentication at diverse websites which support OpenID.

Therefore the user must once create an account at an OpenID provider of his/her choice. When the user wants to log in at a website via OpenID, he/she enters his/her OpenID there. Thereafter the user is redirected to the corresponding OpenID provider where the identity is validated, e.g. by means of a password. The user is also asked whether he/she trusts the original website. If that is the case the user is redirected back to the original website along with the identity data offered by the OpenID provider. The standard identity data are the nick name, email address, full name, date of birth, gender, postcode, country, language and time zone and the OpenID user can select which of these attributes is sent to a website during the authentication process. Now the user is logged in at the original website with his/her OpenID [12].

There is also the possibility to exchange arbitrary identity data. Indeed the OpenID provider and the website using OpenID must both have implemented the *OpenID Attribute Exchange Protocol* [13].

Due to the functionality of OpenID, whereat the user is redirected from the accessed website to an OpenID provider, phishing-attacks are feasible. Thereby an malicious website redirects the user to a tampered OpenID provider to obtain the user's password surreptitiously [14].

2.1.3 Security Assertion Markup Language (SAML)

SAML [15] is an XML-based open standard data format for authentication and authorization. It supports among others single sign-on (SSO) and the centralized user- and authorization management for distributed services. The SAML protocol can be explained as follows:

A subject logs in at an identity provider (IdP), which issues an assertion containing security information about the principal. This security information covers security attributes from the principal and/or permissions to gain access to defined resources or services, respectively.

After the principal has successfully logged in at the IdP, he/she and the respective assertion are redirected to the service provider (SP). In order to ensure the assertion is issued by the trusted IdP, a trusted relationship between IdP and SP, typically based on a public key infrastructure (PKI, see Section 2.1.5) is used. After the assertion has been inspected, the SP knows the identity or security attributes of the principal and can make authorization decisions based on it. The SP can also use the permissions for defined resources, which are contained in the assertion, to grant access to a certain resource.

When a user is authenticated with the aid of an assertion, authorization can be applied to him/her for the current and subsequent operations. Thus SSO is supported by SAML. Further, as the centrally organized assertions may hold permissions for defined services, a centralized authorization management for distributed services is enabled. Indeed the permissions cannot be expressed in a very fine-grained way.

2.1.4 Kerberos

Kerberos [16] is an authentication system which facilitates single sign-on (SSO) without having to transmit passwords or keys. It is composed of an architecture and a protocol. The architecture is composed of clients, services and the Key Distribution Center (KDC) which stores the symmetric keys of each participant. When a client wants to gain access to a service, the communication between the parties makes use of the following protocol:

First the client sends a so-called authenticator to the KDC which happens during the login process. The authenticator contains the user name and parts of the authenticator are encrypted with the user's key which is derived from the user's password. The KDC inspects its database for the user name, verifies the authenticator by decrypting the encrypted part with the user's key and issues a so-called Ticket Granting Ticket (TGT) containing the user's name and IP address, the KDC's name, the client's name and a so-called session key that has an expiry time. The TGT is encrypted with the KDC's key to ensure nobody can alter it. This encrypted TGT and a copy of the session key are encrypted with the client's private key and send back to the client. Thus only the proper client can decrypt this package with its key and obtain the encrypted TGT and the session key.

To gain access to a certain service the client issues an authenticator containing the client's name and IP address and the time and encrypts it with the session key. Then the client sends that encrypted authenticator along with the TGT and the name of the requested service to the KDC, which decrypts the TGT with its key. Then the KDC decrypts the authenticator with the session key obtained from the TGT and checks the content of the TGT against the authenticator's content. This ensures the service request was sent from the legitimate client. Note that the KDC does not have to store the session key, which would be a great effort when there are many users.

A service ticket is issued by the KDC that contains the client's name and IP address, the service's name and a new session key. The KDC encrypts the service ticket with the service's key. Due to this operating mode, nobody can alter this service ticket except the KDC or the service provider itself. The service ticket is only valid for the certain client and the particular service and has an expiry time. The KDC encrypts the encrypted service ticket and the new session key with the first session key and sends it back to the client. The client decrypts the package, takes the new session key, and encrypts a new authenticator with it and sends it along

with the encrypted service ticket to the service provider. Note that only the legitimate client is able to decrypt the package containing the new session key and therefore only the legitimate client is able to encrypt something with the new session key, except the KDC. Furthermore only the KDC is able to encrypt the service ticket with the service's key.

The service provider decrypts the service ticket with its key and next decrypts the new authenticator with the new session key, which was obtained from the service ticket. It checks the authenticator's data against the data contained in the service ticket in order to validate the client's identity. Next the service sends an acknowledgment to the client which has access to the service from now on. Their communication may be encrypted with the new session key.

2.1.5 Transport Layer Security (TLS)

TLS [17] is the successor of Secure Sockets Layer (SSL) and is a standardized protocol, which is mostly used to authenticate a server and establish an encrypted communication between client and server, e.g. a browser and a web server. TLS is based on private-public key pairs and the infrastructure using TLS is called a public key infrastructure (PKI).

Generally, a message is signed by calculating a signature with a private key. The signature is validated with the public key that is associated with the private key. When the public key can be assigned to a certain user, it is ensured that the message signed with the corresponding private key was sent from the certain user as the private key is only known by this user.

It is also possible to encrypt and decrypt messages by means of private-public key pairs. Thereby the message is encrypted with the public key and decrypted with the associated private key. Thus everyone can encrypt a message with the public key but only the intended receiver can decrypt the message with the private key.

The TLS protocol uses public and private keys as follows: When a client connects to a server and sends a request, the server's response contains an X.509 certificate which has been issued and signed by a certification authority (CA). The public key from the CA is known by the client and thus the client can validate whether the X.509 certificate was issued by the legal CA and is trustworthy. The X.509 certificate contains the server's public key and its URL. With the aid of the URL the client can authenticate the server it is connected to. The authentication of the client is also possible and works the same way.

After the authentication has been established, the client generates a secret random number, encrypts it with the server's public key and sends it to the server. Only the server can decrypt this secret number with the aid of its private key. By means of this secret number a symmetric key is created, whereby the traffic between client and server is encrypted. Furthermore, this key is used to calculate a Message Authentication Code (MAC) which ensures that the messages are not tampered during transfer and are sent from the correct party.

2.1.6 eXtensible Access Control Markup Language (XACML)

XACML [18] is a framework to provide a general approach for fine-grained access control. It consists of an architecture, an XML-based language to define policies and another XML-based language to define requests/responses for authorization decisions. Also the mechanism to find the appropriate policy for a request and determine the authorization decision is part of XACML.

In XACML a rule consists of a target, an effect and a condition. The target specifies the resource, the principal and the action. The condition refines the requirements for the applicability of the rule, which cannot be expressed within the target, like the environmental context for example. A rule is applicable when the entire target is equal to the target of the request and the condition holds. An applicable rule permits or denies the action depending on the specified rule effect.

Each attempt to perform an action on a resource is intercepted by the Policy Enforcement Point (PEP), which can be placed on each layer of the IT infrastructure, e.g. business layer or data layer. Then the PEP forms an authorization request mainly based on attributes and sends it to the Policy Decision Point (PDP). The PDP inspects the Policy Retrieval Point (PRP) for the appropriate policy to determine the authorization decision and sends it back to the PEP which acts on it. When there are conditions in the policy requiring additional information, the PDP inspects the Policy Information Point (PIP) for the information. The Policy Administration Point (PAP) is that point where administrators can administrate the policies.

Attributes are primarily used to determine the authorization. A principal, a resource, an action and the environmental context are all modeled with attributes in XACML. Also a request from PEP to PDP is mainly formed by attributes and their values. Consequently the policies consist mainly of attributes and corresponding values. RBAC can be implemented as specialization in XACML.

2.1.7 Evaluation Concerning the Applicability for the Peer Model

OAuth is a protocol to facilitate one application to access defined user resources of another application. This approach is not suitable for the Peer Model (see 3.1). OpenID enables a user to authenticate him/herself and provide his/her security attributes, which would be a fitting approach. However, as OpenID is vulnerable to phishing attacks, this approach is not used. SAML is a data format for defining and transmitting the security attributes of and permissions for an authenticated user. This data format was not used due to its expressiveness and complexity. Kerberos has the advantage of encrypted traffic, but it can only be used in a trusted domain and thus it is not qualified for the deployment in a distributed and untrusted environment, wherein the PeerSpace.NET may be used.

In contrast, TLS is designed for untrusted and distributed environments. However, if TLS were used for authenticating users via their security attributes, they would have to install new certificates every time their security attributes change and revocation lists would have to be maintained. Thus the authentication for users will be achieved with a self-implemented identity provider (see 5.1.3) and TLS will be used to authenticate the identity provider and encrypt the traffic.

The rules of the security model base on the structure of XACML's rules as they are expressive and suitable for distributed systems.

2.2 Authorization Models

In the context of distributed computing, authorization is the process of permitting or denying principals access to a certain resource in a specific way, e.g. read, write, execute or delete,

possibly under particular circumstances. There exist several access control models which are considered next.

2.2.1 Access Control Lists (ACL)

An ACL defines the principals who are allowed to access specific resources in a certain way. Each resource is associated with a list which states the users together with the granted kind of access, which may be read, write, execute or delete. A service administrator has full control over authorization, but the administrative overhead is considerable since the permissions for each resource and each principal have to be set explicitly. [19]

2.2.2 Discretionary Access Control (DAC)

The owner of a resource defines the principals and/or the groups that are allowed to access the resource in a certain way. Thus this approach is similar to ACLs with the main difference that the owner of the resources defines the authorization instead of the service administrator. [19]

2.2.3 Mandatory Access Control (MAC)

Every principal is associated with a security clearance and the resources possess security properties. It depends on these two attributes whether an access is permitted or not. A service administrator has full control over authorization through managing each association. The administrative overhead is not as high as in ACLs, because principals and resources are abstracted. [19]

2.2.4 Role-Based Access Control (RBAC)

In this approach roles are assigned to principals and permissions are assigned to roles which is managed by the service administrator. The mapping from principals to authorization is decoupled like in the MAC approach but RBAC provides a higher expressiveness for access control since there can be more than one role assigned to users. Furthermore assigned roles can be activated or deactivated through the usage of sessions. Each session is assigned to one principal, but there can be more than one sessions assigned to a principal.

Role inheritance is possible which allows the creation of a role hierarchy and delegation of authority. Roles can be mutually exclusive to provide the separation of duties [20] concept. Role A and B are mutually exclusive: either role A or role B can be invoked for a principal, but not both. There is a distinction between static separation of duties (SSD) and dynamic separation of duties (DSD). Static separation of duties is achieved by defining constraints (rules) that allow either role A or role B to be assigned to a principal. In contrast dynamic separation of duties is achieved by defining constraints (rules) that allow either role A or role B to be activated for a principal during a session. The principal is associated with role A and role B but only one role of them can be activated at once. The activation of a rule may also depend on context information, so that the rule is only applicable for a user when a certain condition is satisfied.

RBAC also supports the least privilege concept which means that a principal has only the permissions to perform its task and not more. This is supported because only the least permis-

sions for executing a task can be assigned to a role and only the roles, which are necessary to perform a task, may be activated during a session.

Another strength of RBAC is the capability to model enterprise organizational structures naturally through the deployment of roles and rules. Roles of the security model reflect roles of the enterprise organizational structure that represent competency, authority and responsibility and rules reflect the permissions for the roles dynamically determined depending on conditions. It is also easy to adapt the security model to changes in the enterprise organizational structure because of the encapsulation of permissions and principals through roles and the possibility to define (or redefine) relations between roles, permission and roles, and principal and roles. [21]

2.2.5 Attribute Based Access Control (ABAC)

This access control model works like RBAC with the difference that security attributes of principals instead of roles are used to define permissions. Security attributes are more generic than roles and can be arbitrary properties of the principal, e.g. the name or the gender.

Security attributes are assigned to a principal during the authentication process. Rules define security attributes and specify therewith for which principals they are applicable. This is the case if all security attributes of a rule are a subset of those from the principal. So, e.g. a rule with the security attributes *Species = Human, Gender = Male* is applicable for the principal with the security attributes *Species = Human, Gender = Male, Name = Lukas*, but not the other way round.

Roles can also be stated as security attributes, thus ABAC supports RBAC [22].

2.2.6 Choice of an Access Control Model

ABAC is the most expressive access control model with a moderate administrative effort among the analyzed models. Thus it will be chosen for the security model of the Peer Space.

2.3 Security Mechanisms of Distributed Computer Systems

As authorization and authentication mechanisms have been analyzed, entire security mechanisms from distributed computer systems are investigated next. As the Peer Model is a space-based1 middleware with an underlying P2P architecture, the analyzed systems where chosen according this criteria. So all but one of these models are space-based and/or use a P2P architecture.

2.3.1 Secured Space Services on top of XVSM

At the Vienna University of Technology a space-based system has been developed [23]. The systems itself is not distributed, because a space is hosted by a single computer. Nevertheless, P2P applications can be implemented therewith as spaces which are hosted on different, distributed computers can interact.

Thus the system is capable to achieve load balancing [24] [25] and replication mechanism [26] by its nature.

The implementation of this system is a framework called eXtensible Virtual Shared Memory (XVSM), which has also been developed at the Vienna University of Technology [2].

In XVSM the space is organized in sub-spaces called containers, where entries can be written to and read or taken from, whereby several entries can be handled in a single operation. Thereby XVSM supports several mechanisms, like FIFO, LIFO, key, label etc. to write, read or take entries from or to a container.

A query blocks until it can be satisfied or its timeout expires. Queries may have more than one stage, which are subsequently executed, whereby the output from the previous stage serves as input for the next one. Stages are connected with the pipe operator known from Unix systems [27].

XVSM may be extended with the implementation of additional coordinators. It is also possible to define aspects which are performed before and/or after a space operation, e.g. write. Aspects may use space operations themselves [28].

Clients and services can be brought together in an ad-hoc way with the aid of tuple spaces [29, 30]. Such space-based services can also be established with XVSM by adding request and response containers to the space. By writing a request entry to the request container, the corresponding service is invoked, whose results can be obtained via the response container. The service may use data from arbitrary containers to perform its task [31]. Figure 2.1 illustrates this basic architecture.

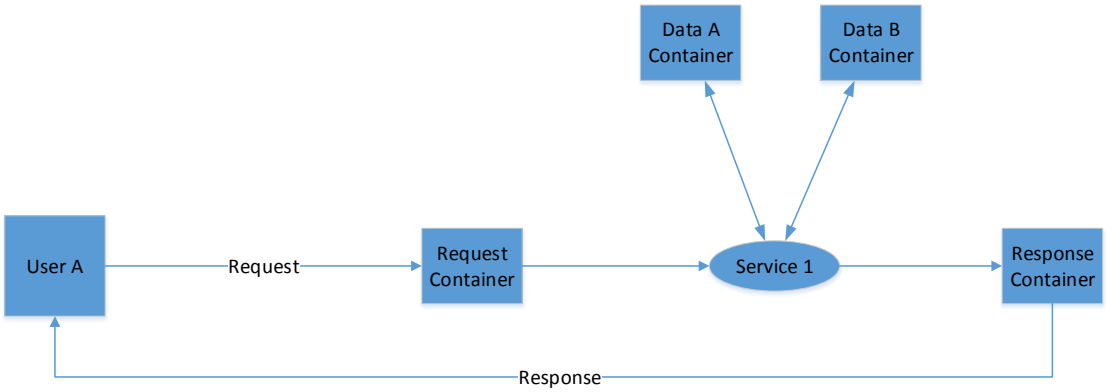

Figure 2.1: Basic space service architecture with XVSM

Authentication

Before clients access the space, they must authenticate themselves by sending credentials to an external authentication provider which issues a cryptographically signed single sign-on (SSO) token that proves the client's identity. The authentication request and the SSO token have to provide at least a unique user identifier. In order to support RBAC the SSO token must also possess roles associated with the particular user. The SSO token is valid for a certain amount of time, thus the client can use it more often than once [31].

Authorization

The policies for authorization are similar to the eXtensible Access Control Markup Language (XACML) [18], which is constituted of an XML-based language for the description of access control policies and a model for its processing. The rules state a target, a scope, a condition and an effect. The target specifies the principal, the kind of access and the targeted resource the rule is valid for. In the context of XVSM the kind of access is either a read, write, or take operation and the resource is a container.

The scope states for which entries, and the condition defines under which circumstances the rule applies. Both are realized with dynamic space queries against proper containers, whereby the scope yields a set of entries, for which the rule is applicable and the condition returns true or false depending whether the queries could be satisfied or not. Dynamic parameters may be used for the definition of the scope field and/or the condition. Thus a rule can be defined which is applicable for entries containing the ID of the sender (scope) under the condition that a registration entry with the sender's ID (condition) is present in a certain container, e.g.

The effect for the XVSM access control model is either permit or deny. A rule applies for an entry when the principal, the kind of access and the container matches, the entry is covered by the scope and the condition evaluates to true. If more than one rule applies for an entry, the resulting effect must be evaluated with the aid of a combination algorithm, like "permit overrides", "deny overrides" or "first applicable".

As in XVSM a write operation may comprise several entries. It is only permitted if access is granted for every entry. Otherwise the whole write operation is denied. Read and take operations may also target several entries, whereby denied entries are ignored and treated like they were not there. Thus it is transparent to the user whether he/she has no access to particular entries or they are not present.

In this access control model all parameters except the effect of a rule are optional, meaning undefined fields are considered like a wild card is stated there. Consequently the creation of general rules is feasible. [32] [28]

Security decisions must be evaluated for every space operation, i.e. when entries are written to or read or taken from a container. Therefore the access manager which conducts this decision is integrated in the XVSM coordination layer, which is responsible for these operations. Thereby the access manager can perform space operations itself, as it must evaluate the scope field and the condition.

Rules are realized as entries and stored in a dedicated space container - the policy container. The policy can be dynamically changed, as rule entries can be written or taken from the policy peer by users with sufficient privileges. Thus the delegation of security administration can be also easily achieved by adding permit rules for particular users, which grant them possibly restricted access to the policy container.

The policy container holds the rules for the access to itself. To grant an user the privileges to administrate the policy, the corresponding rules must be written to the policy container. In order to bootstrap this, the process which starts the space has implicit access rights to the policy container [32].

In order to secure space services against unauthorized access, the direct access to data containers is generally denied. Users are only allowed to access the request and response container

directly and the data containers only via the invoked services. When only access to the request and response container is controlled, a rather coarse-grained access control is achieved. Thereby it can be stated which user is granted to invoke certain services. In order to refine the policy the access to the data containers, used by the services on behalf of the invoking user, can be controlled. Thus the access control for the data containers involves the identity of the direct sender, i.e. the services, as well as the indirect sender, i.e. the invoking user. Figure 2.2 depicts the described interaction of the entities [31].

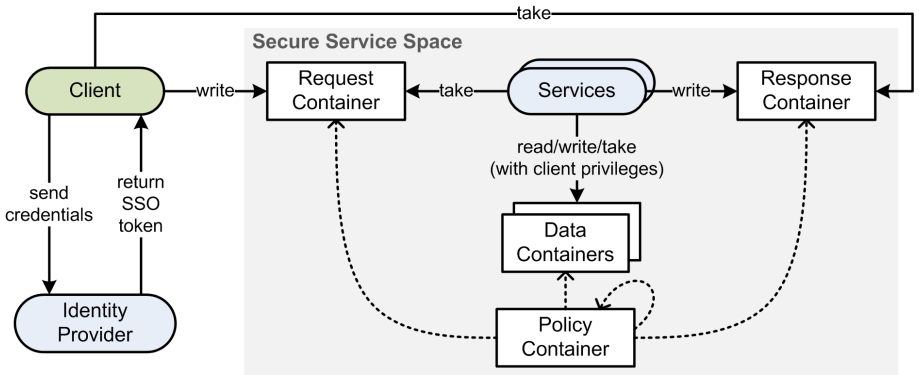

Figure 2.2: Secure service space architecture (taken from [31])

Evaluation and Correlation to the Peer Model

The approach with the SSO token for authentication has the disadvantage in a P2P environment that a malicious space could impersonate a user after he/she has transmitted the token.

The organization of the authorization mechanism is bootstrapped with XVSM mechanisms and the overhead is manageable. Furthermore the administration of the authorization can be delegated to any user and the service's access to a data container can be controlled on the basis of the invoking user. These two attributes leads to a good scalability of this authorization model. It is possible to express fine-grained rules, which may involve attributes from the entries that are access-controlled as well as environmental context information. The access control is transparent to the user, as no difference is revealed whether an operation is not possible or not granted.

The XVSM model with the request and response containers and the service invocations, triggered through sent entries to the request container, is similar to the Peer Model. The request and response containers in the XVSM model are analogous to the Peer Model's PIC and POC, which are more lightweight and do not possess the query capabilities of XVSM containers. In this XVSM model as well as in the Peer Model services are triggered through sending entries to the request container or PIC, respectively. In both models the services may use data from defined containers. In the previous XVSM explanation these containers were called data containers and in the Peer Model they are sub-peers. As in both models the services request data from containers

on behalf of a user, delegation is given. In the XVSM model the delegation chain is two elements long: a service acts on behalf of a user. As in the Peer Model entries may be forwarded from peers or services to other peers and services, the delegation chain may be infinitely long.

As the security model for the XVSM supports authorization for delegation, content-aware rules and the involvement of context information, it fits fairly well to the Peer Model and builds the basis for the Secure Peer Model [9]. This security model supports a sophisticated and highly expressive authorization mechanism which involves, besides content and context information, the identity of the users of a delegation chain as well as information about the authenticating entities. To avoid the disadvantage of SSO tokens in a P2P environment, the authentication mechanism is bootstrapped on an identity provider which verifies the sender's signature and provides his/her security attributes to the receiver for every operation. Thus the sender cannot be impersonated.

The Secure Peer Model [9] is simplified and adapted for the PeerSpace.NET which results in the security model designed and implemented throughout this thesis.

2.3.2 Hermes with RBAC

Hermes is scalable event-based publish/subscribe middleware [33]. Publish/subscribe systems decouple sender and receiver from each other as publishers send data to the middleware and those who subscribed to this kind of data receive it.

The published event is routed to the corresponding subscribers, whereby in Hermes the granularity of subscriptions can be based on event types only or on event types and attributes of the event. The suitable route from a publisher to a subscriber is established as subscribers send a subscription to the middleware and publishers dispatch an advertisement of the event type they are going to post. Subscriptions and advertisements meet at rendezvous nodes and proper routes are established.

Event types are organized in an inheritance hierarchy, so that every published event is an instance of an event type, which possesses an event type name, typed event attributes and an event type owner.

The Hermes network is constituted of event brokers and event clients, whereby event brokers features the publish/subscribe functionality and are connected in the style of P2P among themselves. Clients use the provided functionality and are publisher and/or subscriber. They must be connected to an event broker which is called local event broker. The brokers which are only connected among themselves and not to clients are called intermediate event brokers. When a client connects to an intermediate event broker it becomes an local event broker. Figure 2.3 illustrates the interaction of clients, local and intermediate brokers [34].

Hermes does not provide access control by its own. However, [34] introduces an approach for authorization in Hermes by means of role-based access control rules as described in [35].

Authentication

The clients authenticate themselves by sending credentials to their local event broker and the broker network is trustworthy through the deployment of certificates.

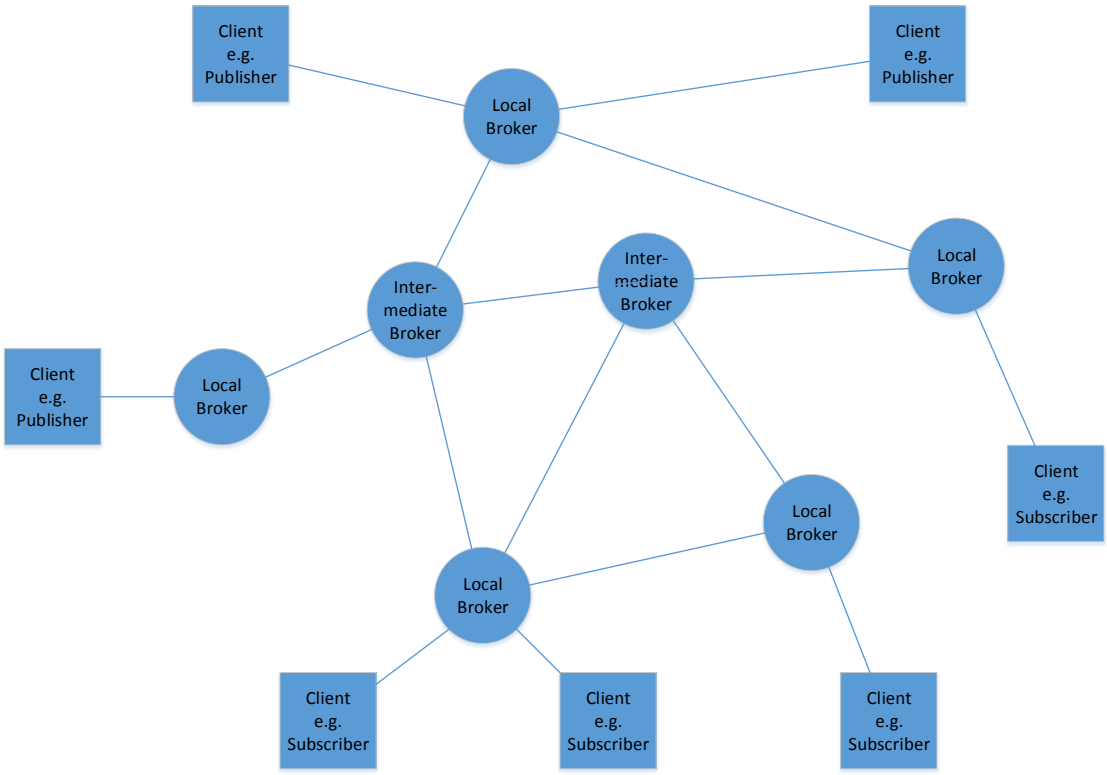

Figure 2.3: Example of a Hermes network

Authorization

The role-based access control is achieved with the aid of different kind of policies. One kind of policies specifies whether a client is allowed to basically connect to its local broker. Another kind of policies defines who is allowed to create, modify and remove a specific event type and the third kind of policies defines the clients who are granted to advertise and subscribe to, respectively, certain event types.

Through restrictions of advertisements and subscriptions for event types, access control concerning sending and receiving events on the granularity of their types is achieved. The restrictions are defined in policies which are located in and enforced by the event brokers. Therefore users send their credentials along with the advertisements and/or subscriptions.

The policy for an event type is created by the event type owner and its distribution is bootstrapped with Hermes. This happens when the event type owner posts a policy evolution event which causes the distribution of the particular policy to the event brokers but not to the clients. Consequently the policies are transparent for the user.

Advertisements and subscriptions can be either be fully granted, partially granted or denied. When they are partially granted restrictions are exerted, which cause in the simplest case, that a sub-type of the desired entry type is advertised or subscribed. The particular sub-type is stated in the event type's policy, each for publisher and subscriber. More fine-grained restrictions op-

erate on the attributes of events. The used attributes are either typed event attributes or arbitrary attributes. Restrictions using the type event attributes have the advantage that the Hermes coordination mechanisms support these kind of attributes and thus Hermes is able to filter events close to the publisher which leads to save traffic and computation in the network. Restrictions using arbitrary attributes of events have the advantage of flexibility but the disadvantage of causing more overhead in the network.

Evaluation

As the policies are created by the particular event type owner, the scalability regarding the security administration is not affected. The traffic overhead depends on the quantity of policies with restrictions using arbitrary attributes. These kind of restrictions are very flexible concerning the used data for access control decisions. The policies as well as restrictions are transparent to the user which is a desirable security approach.

2.3.3 SMEPP Implementation with SecureLime

The Secure middleware for Embedded Peer-To-Peer Systems (SMEPP) [36–38] is a service-oriented P2P model with access control capabilities. The model is constituted of peers that interact among themselves. They are service provider and/or service consumer. Thereby synchronous, asynchronous and event-based admissions to services are supported. For a synchronous service operation the invoker blocks until the particular operation has terminated. In contrast the invoker of an asynchronous service operation waits only until it has started. It is also possible to participate in a service operation by means of events, whereby the invoker of an event does not block until the particular operation has been posted, but it is invoked as soon as it is available. Thus events can be seen as non-blocking invocations of service operations.

The access control is achieved with the aid of groups, which are logical associations of peers. Only peers within a group can host and consume service operations and events from each other. Therefore a peer can join a group, whereby a peer sends its credentials along with the join request. When the credentials are valid for the group, the peer is allowed to join the group. When a peer is a member of a group it can host services for this group, which can be accessed by all group members.

A peer may join several groups in order to provide or consume services to/from particular groups. Services may consume other services, but they are not granted to join groups on their own. A service can be invoked via the group they are posted in or by means of the hosting peer, but the peer providing and consuming the operation must reside in the same group anyhow. Figure 2.4 illustrates an example where Peer A and B provide their services in Group 1 and Peer E hosts its service in Group 2. Only peers that are members from the particular group can access the hosted services. So Peer D does not provide any service, but it may access Service 6, which is provided by Peer E, and Peer C's Service 4. Peer C also hosts Service 5 in Group 1. Since Peer C is member of both groups it is allowed to access all depicted services [37].

The paper [36] introduces an implementation of SMEPP with SecureLime [39], which extends the federated tuple space language Lime [40] with security features.

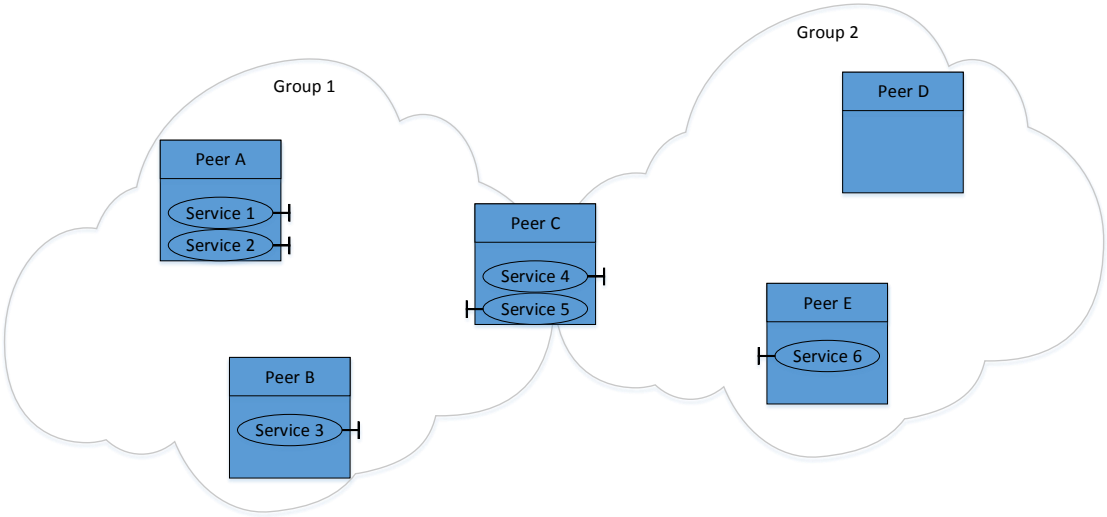

Figure 2.4: Example of groups and hosted services in the SMEPP model

Authentication and Access Control

According to the SMEPP model peers possesses an AppKey and passwords for groups. A peer with a valid AppKey is permitted to join the SMEPP application and must provide a valid group password to get access to the particular group.

The SMEPP primitives of group and service discovery are realized with SecureLime via two tuple spaces holding the group and service descriptions, respectively. They can be discovered by reading the particular tuple space. A peer creates a group and registers it by creating such a group description tuple space and writing the particular group description to it.

The group description tuple spaces of all peers providing the valid group password are merged and thus only legal group members can communicate and perform the SMEPP primitives via this space. The group description spaces from peers with invalid passwords are not merged and thus these users cannot join the group. The password validation and the merging is performed by SecureLime and the traffic is encrypted using the passwords [36].

Evaluation

The authentication is achieved via passwords which grant access to groups. Thus authentication is restricted to groups, which limits the granularity for access control as it does not involve any content or context information. As groups can be autonomously created and joined by peers the scalability is not affected by the access control mechanism. As peers can not see groups and services they are not allowed to join and access, the access control is transparent to the user.

2.3.4 Tuple Centers Spread over the Network

The Tuple Centers Spread over the Network (TuCSoN) [41] model is a multi-agent system for the coordination of distributed processes, as well as autonomous and mobile agents. The basis

of TuCSoN are tuple centers which are the well-known tuple spaces [42] with the enhancement of programmable reactions to space operations. Thus when an agent writes, reads or takes a tuple to/from a tuple center via Linda primitives, a reaction of the tuple center is triggered. Such a reaction can be calculations and further space operations, so services as well as dynamic coordination logic can be realized therewith. A reaction is carried out transactionally, meaning either all operations and calculations, directly and indirectly triggered, are successfully conducted or none.

Tuple centers are collected in nodes, which in turn are composed in domains. A domain possesses a gateway node which holds administration tuple centers. The place nodes, which contains the application tuple centers, are accessible from outside the domain via the gateway node. A place node may by a gateway node itself which leads to nested domains, whereby hierarchies can be established [43].

Figure 2.5 depicts the correlation between tuple centers, place nodes, administration tuple centers, gateway nodes and domains by an example.

Authentication

The authentication mechanism depends on the particular implementation of the TuCSoN model. The gateway node may execute the authentication.

Authorization

A gateway node controls the visibility and the authorization of the nodes and sub-domains which reside in its domain. Thus the enforcement of policies from place nodes and sub-domains must be delegated to the parent gateway node. The mechanism used by the gateway node is bootstrapped with the concept of tuple centers. Thus the security administrative tasks can be conducted with the tuple centers.

The paper [44] introduces a role-based access control model for TuCSoN which is naturally adaptable to the concept of agents and the agent coordination context (ACC) by which the agent interacts with the system. The authors argue that an agent in TuCSoN is similar to a principal in the RBAC model. The agent negotiates an ACC, whereby it gains access to the particular domain. The ACC is dynamically negotiated and depends on the identity of the agent or its role affiliation and the domain that is going to be accessed. The ACC may depend on the context of the agent itself, e.g. whether another particular ACC is active for the agent. So separation of duties can be achieved and the ACC is similar to a session in the RBAC model.

In the introduced access control model for TuCSoN the role policies are stored in tuple centers which leads to the possibility of viewing and changing them dynamically. The policies consist of permit rules specifying a role, a previous role state, a later role state and a condition. In this model a role can have several role states. A rule applies when its previous role state matches the role state of the user who triggered the operation. After the operations has been executed the user's role state changes to the later role state which is specified in the rule granting the operation. This concept facilitates the enforcement of workflows [44].

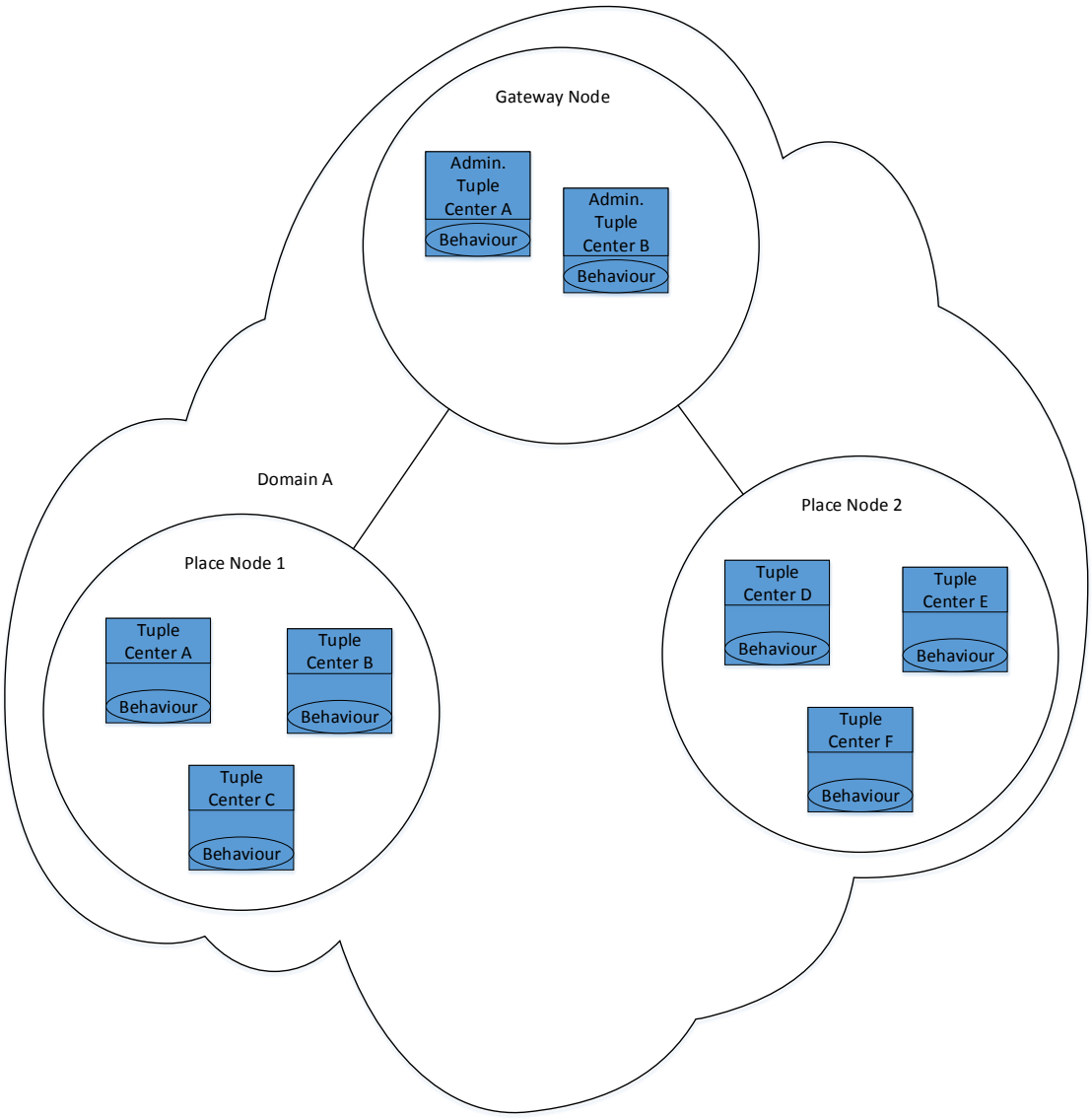

Figure 2.5: Example of a domain in the TuCSoN model

Evaluation

The access control is bootstrapped with the TuCSoN model itself and thus the overhead is minor. Further it is possible to change the policy dynamically with mechanisms featured by the model, as rules reside in an administration tuple center. Since the coordination and the access control are achieved in the same place, namely the gateway node, the authorization is transparent to the user or agent, respectively. For him/her there is no difference whether a certain place node is not there or not granted for access, because in both cases it is invisible for the user.

There is one disadvantage concerning scalability, as the gateway node enforces all policies

for the entire domain. Therefore the paper [43] mentions an optimization approach, namely the gateway node should only enforce the policies of its sub-domains on rather coarse-grained access control lists (ACL) and the more fine-grained access control is conducted by the particular sub-domain gateway node.

Fine-grained rules can be established, which state roles, conditions and even role states. These possibilities are supported by the coordination capabilities of TuCSoN's tuple centers.

2.3.5 Windows Communication Foundation (WCF)

The WCF [45] is no P2P-like coordination middleware like the previously analyzed systems, but it is comparable to them since it also uses messages as described below. Besides, it has a very broad and flexible field of application.

It is a framework shipped with .NET to build and employ remote services and thus to create programs corresponding to the service-oriented architecture (SOA) paradigm. Thereby the invocation and response of services are conducted by means of message exchanges, which may occur according to several patterns. For a synchronous service invocation, whereby the calling program blocks until it receives an answer, the request/response pattern is established. Asynchronous service invocations, where no reply is expected, are performed with the aid of one-way messages.

Generally service, client and provider are loosely coupled and thus interoperability is given, as long as the service contract holds. This contract determines the parameters and the return type of the service, which must be known in order to invoke it. For this reason WCF derives the service meta data from the contract and provides it to the client, which uses this data to create a proxy object. Then the client calls the service method on the proxy object and thus invokes the service.

In WCF the application logic of the service is decoupled from its hosting and the way it is accessed. Therefore WCF follows the ABC principal, which stands for address, binding and contract. The address is an URI that specifies where the particular service can be accessed and the binding defines how data is exchanged, i.e. the communication protocols and message encoding. Address and binding constitute the endpoint of a WCF service. Data in form of messages is sent to endpoints and thus the particular service is invoked. Analogously the returned value is sent as message to the client's endpoint.

Services are either hosted by the internet information services (IIS) platform [46] or an application. The binding comprises SOAP, text and binary format encoding and protocols like HTTP, TCP Named Pipes and MSMQ. Thus WCF is capable to establish a reliable and durable message exchange. Furthermore WCF supports several security features, like authentication, authorization as well as transport and message encryption [45, 47].

Authentication

WCF provides authentication via certificates or user/password pairs and supports the Windows authentication.

The supported Windows authentication mechanisms are the Kerberos protocol [16] and the NT Lan Manager (NTLM) [48]. For authentication via certificates the standardized X.509 certificates [49] are deployed.

The user/password approach bases on ASP.NET [50] which associates users with roles and thus supports RBAC. The Identity Model allows to define security attributes additional to roles, thus ABAC is supported.

In WCF services can also be authenticated in order to ensure the service is not spoofed [51–53].

Authorization

The invocation of service methods can be controlled by means of the membership in Windows Groups, ASP.NET roles or X.509 Certificates. Further security attributes from the invoker may be used to evaluate a security decision.

Access to resources is granted by means of claims, which are dynamically assigned to users based on their identity and the security policy specified for the resource. Thereby it is possible to state a dependency between the assignment of claims and the presence and/or absence of other claims. Thus separation of duties can be achieved. When a service requires access to a certain resource, access control concerning the invoking user and this resource can be defined via so-called claims. The Authorization Manager [54] conducts the assembling of the applicable claims, which results in the authorization context, which in turn is used to perform access control to resources. The authorization manager also facilitates access control to specified operations, which are summarized in tasks.

When a client calls a service, which needs to invoke another service for its execution, the latter one is also called on behalf of the client. In order to perform sufficient access control on the indirectly called service, the identity of the original invoker must be forwarded. For that reason the service-calling service impersonates the original client. [51–53]

Evaluation

The security features in WCF are not bootstrapped with the framework itself, but instead proven solutions may be established for the particular field of application. The caused overhead depends on the deployed modules, but is manageable since they are dedicated solutions. When access to a service is not granted, an access denied error message is returned. Thus the access control is not transparent to the user. Generally the scalability of WCF services depends on diverse settings and used patterns.

There is a broad field of authentication mechanisms and thus RBAC and ABAC are supported. The granularity of authorization can be considered as fine-grained, since the invocation of services as well as the access to resources can be controlled. Further dynamic permissions can be established with the aid of claims and delegation is also possible via impersonation, although the services must be trusted therefore. Policies can be defined and changed dynamically in the authorization manager.

2.3.6 Comparison of the Security Features

We conclude the related work section with Table 2.1, which compares important security features of the analyzed systems. Content-aware rules use data from the transmitted message (for write operations) or information about the queried data (for read/take operations) to specify whether the rule is applicable. The context-aware rules incorporate context information from the middleware. When a service calls another service on behalf of a user, access control that depends on the participants from this delegation chain may be useful. As the service invocations are conducted by means of sent messages, the concerning security feature is called authorization for indirect sender. The wildcard support for indirect sender defines the capability of specifying wildcards which are valid for one or arbitrary elements of the delegation chain. Dynamic policies means that access control rules can be added or removed during runtime. Administration delegation is provided when selected users can be authorized for the security administration. Bootstrapped architecture implies that the security mechanisms are bootstrapped with those of the protected system. Transparency means that the authorization system does not divulge any information, e.g. about denied operations.

It turned out that the realizable and useful security features heavily depend on the middleware they are applied to.

	XVSM	**Hermes**	**SMEPP**	**TuCSoN**	**WCF**
RBAC	$+$	$+$	$-$	$+$	$+$
ABAC	$+$	$-$	$-$	$-$	$+$
Content-aware rules	$+$	$+$	$-$	$+$	$\sim$
Context-aware rules	$+$	$+$	$-$	$+$	$\sim$
Authorization for indirect sender	$\sim$	$-$	$-$	$\simeq$	$\simeq$
Wildcard support for indirect sender	$-$	$-$	$-$	$-$	$-$
Dynamic policies	$+$	$+$	$-$	$+$	$+$
Remote policy changes	$+$	$+$	$-$	$+$	$+$
Administration delegation	$+$	$+$	$-$	$+$	$+$
Bootstrapped architecture	$+$	$+$	$\sim$	$+$	$-$
Transparency	$+$	$+$	$+$	$+$	$-$
Scalability	$\sim$	$+$	$\sim$	$\sim$	$+$

Table 2.1: Comparison of the security features from the analyzed systems

$+$: supported
$\sim$: supported with limitations
$\simeq$: supported with major limitations
$-$: not supported

Explanation to Some Ratings

For WCF's field "Content-Aware Rules" the rating is stated with "$\sim$" because on the one hand the Authorization Manager supports no rules which depend on the arguments with which a user

calls a service, but on the other hand the Authorization Manager supports rules that specify the methods a certain user is allowed to call. Thus access to the returned data types and values can be controlled. "Context-Aware Rules" is rated with "$\sim$" at WCF because context can only be involved in the access control by means of claims which rather state a user context than environmental context.

When the gateway nodes in TuCSoN have to enforce the access control for a huge domain, they may also be a bottleneck. At XVSM there is no bottleneck concerning the security and thus the rating for scalability is "+" likewise in Hermes. As the synchronization in LIME does not scale well and SMEPP is built on LIME, the scalability rating for SMEPP is "$\sim$".

"Authorization for Indirect Sender" is supported by XVSM, but only for one indirect sender (i.e. service for user). Thus the respective rating is "$\sim$". The TuCSoN model has no concrete mechanism for authorizing indirect senders. However, due to TuCSoN's coordination capability, authorization for indirect senders could be implemented in this model and thus the rating for "Authorization for Indirect Sender" is "$\simeq$". The corresponding rating for WCF is "$\simeq$" as impersonating the indirect sender is not expressive.

Findings for the Design of the Security Model

As analyzed in section 2.2.5, ABAC is an very expressive access control model and thus it is chosen for the PeerSpace.NET's security model. Context- and content-aware rules are useful, since a fine-grained policy can be established therewith. Authorization for indirect Sender is a logical consequence due to the PeerSpace.NET's capability to forward entries. The ability to change the policy dynamically is beneficial, as it is annoying and may be expensive to restart a running system when the policy is adapted. For the sake of practicability the potential to administrate the policy remotely via the middleware and possibly delegate someone with the policy administration is handy.

To bootstrap the security model with the protected coordination system itself is a natural choice since the distribution of rules is a coordinative task. Further the dynamic transmission of entries and rules supports dynamic policies. Transparency is desirable because no information about the policy should be exposed. As one of the main goals of P2P systems is scalability, the security must not affect this property. Scalability is supported due to the fact that the security is bootstrapped with the scalable system itself and no extra entity is introduced which may be a bottleneck when the system scales.

Every security feature from the table will be included in the design for the developed security model.

CHAPTER 3

Background

While the related work serves for orientation and decision guidance, the background section's use is to familiarize with the Peer Model, the PeerSpace.NET and the sophisticated Secure Peer Model which serves as basis for the security model designed and implemented throughout this thesis. The used technologies are also summarized in this chapter.

3.1 Peer Model

The Peer Model, explained in [3, 9], is a space-based data-driven coordination middleware for distributed environments. It strictly separates the coordination from the application logic, which results in a good maintainability as well as the possibility to reuse proven coordination patterns.

The Peer Model is composed of loosely coupled *peers*, which represent addressable autonomous units. Data is wrapped into entries and asynchronously transmitted between peers in the style of asynchronous message queues or tuple spaces. Subsequently arbitrarily defined services are invoked by means of the transmitted entries [3].

3.1.1 Interaction of Peers, Wirings, Entries, Containers and Services

A peer consists of two containers, which are similar to those of XVSM but more lightweight as described in Section 2.3.1. A peer's containers serve for input and output and are consequently called *Peer Input Container (PIC)* and *Peer Output Container (POC)*. Further peers possess *Wirings* which read or take entries via their guard links from a source container, call services and write entries via their action links to a destination container. Figure 3.1 depicts the interaction of peers, wirings, entries, containers and services. Thereby Wiring A moves two entries from Peer A's POC to Peer B's PIC. The Wiring B takes one entry from Peer B's PIC and hands it to the Service B which performs some calculations on the entry and finally Wiring B writes this entry to Peer B's POC.

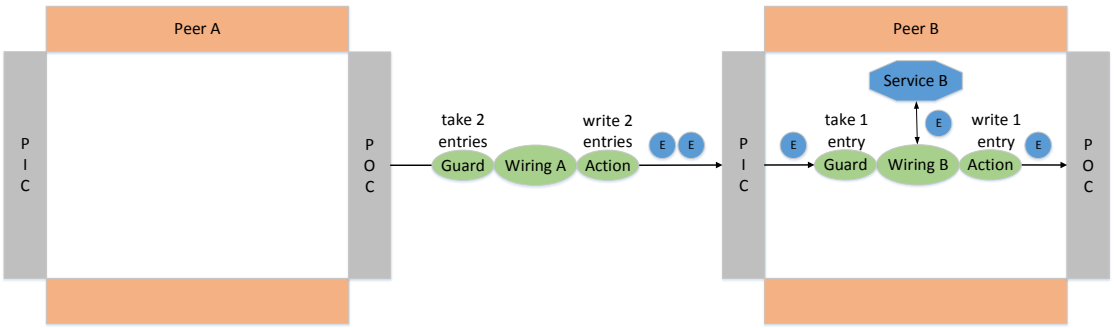

Figure 3.1: Interaction of peers, wirings, entries, containers and services

Peers

There are several kind of peers performing different tasks. Peers may be nested which leads to a hierarchy of arbitrary depth.

Application Peer (APP)
The application logic realized as service methods is processed in an APP, so the application peer is a peer in the classical meaning of this model like the Peer B depicted in figure 3.1.

Space Peer (SPA)
The SPA is like a container and its only task is to store entries. This may support communication and synchronization of concurrent threads.

Coordination Peer (COP)
The COP performs only coordination logic like lookup, routing, filtering, etc.

Runtime Peer (RTP)
The environment for all hosted peers are bootstrapped with the enclosing RTP, whose name corresponds to the local site.

Containers

Containers constitute the spaces in the Peer Model and represent a peer's input and output. They provide a put operation for writing entries into it and a get operation to read or take entries from it. All operations are transactional.

Entries

Requests, responses and sent data are encapsulated in entries which consist of a payload and meta information, called application and coordination data, respectively. While the application data represents the actual data, the coordination data is constituted of labeled properties which serve for coordination purpose, i.e. are used by the queries from action and guard links. Users can also add coordination data, i.e. labeled properties, to an entry.

Entry type
: The entry type is explicitly set by the user and does not have to be the type of the wrapped application data. The entry's type is an essential part of a query against a container and is mandatory. All other listed coordination data is optional.

Time-to-start (TTS)
: With the aid of the TTS the start of an entry's life can be delayed. Wirings ignore entries which have not started their life time yet.

Time-to-live (TTL)
: With the aid of the TTL an entry's life time can be limited. When it is expired, it will be ignored by the wirings.

Flow identifier (FlowID)
: By means of the FlowID, the Peer Model has the ability to correlate associated entries belonging to the same (work)flow, so that wirings take only entries which belong together by means of the FlowID.

Destination property (DEST)
: A service can set an entry's DEST and thus it is sent to the desired destination, i.e. peer. This is conducted by the Peer Model and no wiring to the targeted peer needs to exist.

Wirings

A wiring reads or takes *Entries* via specified *Guard Links* from a source container, writes them into an internal container (*Entry Collection*), performs *Services* on the entries and writes them, or newly generated entries, to destination containers via defined *Action Links*. Thereby the source and the destination containers are the peer's or a sub-peer's PIC or POC and the entry collection resides in the wiring. The guard links are an aggregation of queries against a specified container. The action links are an aggregation of queries against the entry collection and possess specified target container.

A query specifies the entry's coordination data, e.g. the type and the FlowID, an amount, the relational operator and the operation type (i.e. read or take). A query is defined against a container and executed when the query can be satisfied. For example, a container includes seven entries of the type *string* with the *FlowID* "123" and is queried with a take operation which states *entry type = string, FlowID = 123, amount = 2, relational operator = more than*. The query is satisfied as more than two entries of type *string* with the *FlowID* "123" reside in the container and thus the take operation is executed and all seven entries of type *string* with the *FlowID* "123" are taken as the semantics is to read/take as many entries as possible.

The guard links are blocking queries against the specified container. When every guard link can be fulfilled the queries are executed and the wiring fires. Thereby all queried entries are written to the entry collection. In order to prohibit that the wiring fires infinitely, at least one of the guard links must be a take operation.

After the wiring has fired, all services specified by the wiring, are subsequently called. Services are optional and are realized as arbitrary methods. They can use entries from the entry collection as arguments and may emit new entries to the entry collection.

After all services are executed, the optional action links write entries from the entry collection to a specified destination. The action links are queries against the entry collection. In contrast to the guard links the action links are non-blocking queries which means that infeasible action links are skipped. Finally all remaining entries from the entry collection are dropped.

The guard and action links represent the coordination logic and the called services constitute the application logic. They are completely decoupled from each other and are brought together by the wiring. Peers are also connected via wirings among themselves. The same wiring can be run several times in parallel, which allows concurrent executions of the same service.

3.2 PeerSpace.NET

The PeerSpace.NET is a Peer Model implementation which has been made in the context of a master thesis at the Vienna University of Technology [4]. Some restrictions apply for the PeerSpace.NET which are summarized in the subsection 3.2.1.

In the PeerSpace.NET peers are entirely implemented as *Application Peers*, which may act as *Runtime Peer* or as *Sub-Peers*. Coordination and space peers are not specifically implemented as they can be realized with application peers: a coordination peer is an application peer possessing wirings without services and an application peer's PIC can be used as space peer.

Runtime peers possess an I/O component in order to transmit entries to other runtime peers, which are possibly located remotely. The I/O component is implemented via the *XcoAppSpaceCommunicationLayer* and is bootstrapped with the *XcoAppSpace* [55].

Sub-peers do not have their own I/O component, must reside in a runtime peer and cannot own other sub-peers. They cannot directly send entries to an exterior runtime peer. Thus a runtime peer's wirings must convey the entries from sub-peers to its PIC or POC in order to sent them to a remote destination. Sub-peers also cannot be addressed remotely, so entries can only be sent to the encasing runtime peer, which forwards the entries to its sub-peers according to the stated wirings.

There are three kinds of wirings, namely intra-peer, inter-peer and dynamic wirings. Intra-peer wirings connect the containers within a runtime peer, i.e. its PIC, POC and sub-peers. Inter-peer wirings connect runtime peers among themselves, whereby the destination container is always the PIC of the receiving runtime peer. Dynamic wirings can be intra- or inter-peer wirings which are added during runtime. It is subsequently checked whether the corresponding guard links can be satisfied. Wirings can only be added or removed with a reference to the corresponding peer object, i.e. not remotely.

The transfer of entries between runtime peers is realized via inter-peer wirings which reside on the sender side. Thus entries cannot be taken or read by means of a remote peer or wiring. Rather the wiring from that peer, where the entries reside, takes or reads them and sends them to another runtime peer.

Every entry possesses in its coordination data a *destination property (DEST)*, which can be set by services. When the DEST property is set in an entry, it is sent to the destination according to the DEST property before the wiring's action is executed. The DEST can only be set by a service that belongs to a runtime peer's wiring and the specified destination can also be only a runtime peer. Thus, with the DEST property entries can only be sent between runtime peers.

A runtime peer is instantiated by one process and sub-peers and wirings can only be added with a reference to the particular runtime peer. Consequently there is only one user per runtime (peer).

3.2.1 Restrictions of the PeerSpace.NET

No nested sub-peers
In the PeerSpace.NET sub-peers cannot possess sub-peers which leads to a maximum of two hierarchy levels.

DEST property restriction
The DEST property can only be used to sent entries between runtime peers. Sending to sub-peers with the aid of the DEST property is not supported.

No FlowID support
The concept of a flow is not implemented in the PeerSpace.NET.

One runtime user
There is only one user per runtime (peer).

Inter-peer wirings
Inter-peer wirings always reside on the side of the sending runtime peer and can only be added by the corresponding user.

3.3 Secure Peer Model

The Secure Peer Model [9] is a very expressive security model for the Peer Model. The main features of the Secure Peer Model are fine-grained rules and a sophisticated attribute-based access control for (meta) operations in the Peer Model as well as the capability to involve the identity of indirect senders for access control decisions. The Secure Peer Model is based on XVSM access control as described in Section 2.3.1.

The fine-grained rules may define criteria for which entries the rule is applicable in the specified containers. These criteria are defined in the scope field which specifies the entry type and values for fields of the entry. These values can either be static, e.g. *ID=123* or dynamic,e.g. *ID=ID of the sender*.

Context information, which is modeled with arbitrary entries in defined containers within the local peer, can also be stated in a rule. This rule is applicable if the condition, which is a query against the containers holding the context information, is fulfilled. The condition may also use dynamic values similar to the scope field.

In the Peer Model it is possible to forward entries, or emit entries on behalf of other users. Thus for entries there are direct and possibly indirect senders. The Secure Peer Space is capable to perform access control on direct and optionally indirect sender. When an entry is sent from user A's peer to user B's peer and subsequently forwarded to user C's peer, access control at user C's peer can be enforced which involves the identities of user A (indirect sender) and of user B (direct sender). The according delegation chain is (*User B* **for** *User A*).

In this example user C's peer authenticates user B's peer and user B's peer authenticates user A's peer. Thus user A is authenticated at user B and user B is authenticated at user C. The associated authentication chain is (*User A @ User B @ User C*). When user C performs access control including the identities of user A, user C must trust user B to have correctly authenticated user A. In this case the authentication chain is like the delegation chain in reverse order. This is not necessarily the case.

A meta model of the Peer Model was introduced where wirings and sub-peers are organized with special entries in meta containers which leads to the possibility of adding or removing wirings or sub-peers to/from a peer. Thus it is possible that user A adds a sub-peer to user B's runtime peer. When user A's sub-peer sends an entry to user C's peer via the DEST property, the delegation chain is (*User A*) as user A sent the entry directly to user C. As user B's runtime peer performs the sending, user B's peer authenticates at user C's peer and claims that user A's sub-peer is authenticated at user B's peer. Thus the authentication chain is (*User A @ User B @ User C*), which is different from the delegation chain. Thus an authentication chain has to be specified for every principal separate from the delegation chain. The resulting data structure is called subject tree. With the aid of the subject tree trust for senders and the corresponding authenticators can be expressed. A rule defines subject templates which are matched against the subject tree. The senders in the subject template are specified with security attributes and respective values. A sender of the subject tree matches a sender of the subject template when the security attributes and values from the subject template's sender are a subset of the security attributes and values from the subject tree's sender. A subject tree matches a subject template when all senders match in the correct order and the corresponding authentication chains match too. The subject template supports wildcards for senders and authentication nodes.

In the Secure Peer Model rule can be expressed for write, read and take operations to peers. Further the Secure Peer Model facilitates to define rules on meta container which enables to control who is allowed to add or remove specified sub-peers or wirings. The same applies to the policy container resulting that the access to the policy is secured by the policy itself.

The Secured Peer Model builds the basis for the developed security model for the PeerSpace.NET.

3.4 Used Technologies

As the Peer Model and PeerSpace.NET have already been studied, the next step is to depict the used technologies which are the .NET Framework [56], C# [57], Visual Studio with ReSharper [58] and Log4Net [59]. WCF has also been used to employ the identity provider as described in Section 6.2.7.

.NET is a software framework developed by Microsoft which consists of the class library and the common language runtime CLR, which constitutes an intermediate layer between .NET programs and the executing computer.

C# is a object-oriented programing language, made by Microsoft in order to develop .NET programs.

As the .NET framework with the language C# has been used, we decided to program with Microsoft's integrated development environment (IDE) Visual Studio. In order to support a good

programming style and usability, ReSharper from JetBrains has been deployed.

Log4Net is a logging library by Apache for .NET. It facilitates the logging with different log levels and several outputs, like the console, a text file, a database and many more. The logging is important for debugging purposes on the one hand and for recording unauthorized access attempts on the other hand.

CHAPTER 4

Requirements

Before illustrating the design of the security model, it is very useful and a good approach to define its requirements. They derive partly from other P2P-like middlewares considered in the related work section and from the Peer Model, discussed in the background chapter. As the developed security model is designed for and implemented in the PeerSpace.NET, its restrictions affect the requirements. Some requirements are derived from the security model for the eXtensible Virtual Shared Memory (XVSM) and from the Secure Peer Model.

Following all gathered requirements are listed and briefly described in the context of the PeerSpace.NET. They are classified in functional and non-functional requirements to provide a clear representation.

4.1 Functional Requirements

Functional requirements specify a certain behavior of the system.

Usage of attribute-based access control (ABAC)
When the system enlarges, privileges of users or rules changes, the security policy should still be administrable. Therefore rules will not be directly mapped to users, instead security attributes will be mapped to users and rules will be linked to security attributes. This mechanism is called ABAC and commonly used as described in section 2.2.5.

Trusted user administration
The Peer Space is a distributed, P2P-like coordination middleware, where no centralized server for e.g. user administration exists. Peers have to know and rely on the sender's security attributes in order to enforce access control, but no mutual trust can be assumed. External trusted entities which administrate and verify users must be introduced. It is assumed that these entities are handled by *one* organization.

Access-controlled resources
Rules will constrain the access to the containers within the Peer Space, i.e. the PICs and POCs.

Expressive rules
Rules shall be fine-grained and expressive. Thus they will be content- and context-aware with the ability to also specify dynamic content restrictions.

Wildcard support
In order to facilitate the expression of generally applicable rules, wildcards for specific rule sections will be supported. Thereby it is possible to define general rules which are valid for e.g. all entry types.

Permit rules only
Most use cases are realizable with the aid of permit rules. In order to keep the readability of policies high and the evaluation of rules simple, the access control model shall get along with permit rules only.

Support for delegation
In the Peer Model it is a common scenario that entries are sent on behalf of other peers. Therefore the introduced security model has the ability to identify delegated entries by means of all of its senders, rather than to enable the impersonation of the original sender, as no mutual trust can be assumed.

Every peer possesses its own security policy
As the PeerSpace.NET is a coordination middleware in the style of P2P networks, whereby no centralized server exists, every peer enforces its own security policy.

Management of rules is bootstrapped
In order to manage the rules no additional mechanism should be introduced, rather the Peer Space's own functions will be used for this purpose.

The security policy can be changed during run time
When a peer's security policy changes, it should not be necessary to restart the particular peer in order to enforce the new policy.

Setting the security policy remotely
This requirement is useful when the administration of the security is delegated to another user or when someone wants to change his/her security remotely without using a separate remoting tool.

Enforcement of access control is optional
It shall be possible to program a Peer Space successively, i.e. without access control in the first step. Thus the functionality of the Peer Space can be set up and tested without the distraction of security mechanisms. Afterwards the Peer Space can be programmed with enabled security by changing a little setting in the peers' configurations.

4.2 Non-Functional Requirements

These requirements refer to characteristics rather than a special behavior and do not differ from (security) requirements of other computer systems.

Integrity and authenticity
In order to hamper the possibility to inject malicious entries, the integrity and authenticity of the transferred entries must be ensured.

Confidentiality
As the PeerSpace.NET can be deployed in an open environment, e.g. the internet, and the transferred data may be sensitive, confidentiality must be ensured.

Scalability
As it is in the nature of P2P-like systems to grow, the security mechanisms of the PeerSpace.NET must fulfill this requirement.

Usability
As the most sophisticated system is worthless if it is not employable due to bad usability, this requirement is also important for security systems.

Maintainability
When policies change or users are added/removed the effort to adapt the security model should not be major.

Performance
In order not to hamper the practicability of the secured PeerSpace.NET, its security mechanism must not slow down the Peer Space drastically.

CHAPTER 5

Design

The Secure Peer Model [9] is a sophisticated security model for the Peer Model. The PeerSpace.NET is an implementation of the Peer Model. Thus the Secure Peer Model [9] builds the basis for the security model of the PeerSpace.NET. Due to the restrictions of the PeerSpace.NET, its security model differs in some points from the Secure Peer Model [9] as described in the following.

As in the PeerSpace.NET no nested sub-peers are feasible, the developed security model does not have to deal with hierarchies. Further the PeerSpace.NET does not support the Peer Model's concept of a FlowId. Thus, the security model does not have to consider FlowIds.

There is only one user per runtime peer in the PeerSpace.NET, i.e. all wirings, services and sub-peers within a runtime peer belong to one user. So this user is responsible when entries are sent from the runtime peer for whatever reason. When these entries are received from another runtime peer, it must only authenticate the user of the sending runtime peer for security reasons. When these entries are forwarded to a third runtime peer, this runtime peer must only authenticate the second runtime peer and so forth.

For example, user A's runtime peer sends entries to user B's runtime peer, which forwards them to user C's runtime peer. The corresponding delegation chain is (*User B* **for** *User A*), which means user C's runtime peer receives entries from user B's runtime peer, which in turn sent the entries on behalf of user A's runtime peer.

The authentication chain in this example is as follows: (*User A* @ *User B*). User C can be omitted in the authentication chain for access control as the last element in this chain is always the own runtime user. Thus the authentication chain is like the delegation chain in reverse order. As the PeerSpace.NET does not support a meta model and thus only one user per runtime peer is possible, the delegation chain is always the authentication chain in reverse order in the secure PeerSpace.NET. Thus, in the security model for the PeerSpace.NET, specifying a delegation chain is sufficient to perform access control on direct and indirect senders. There is no need to extra specify by whom the user has been authenticated.

In contrast, the Secure Peer Model [9] introduces a subject tree, which defines the authentication chain for each principal of the delegation chain. This is necessary because the Secure

Peer Model is designed for a Peer Model where a runtime peer may possess sub-peers or wirings which belong to another user than the runtime user as explained in Section (see 3.3).

For the secure PeerSpace.NET we assume that all runtime peer users are administrated by one organization.

In the PeerSpace.NET wirings cannot be added from remote and wirings between runtime peers reside always on the sender side. Thus when entries are transmitted from runtime peer A to runtime peer B, the entries are written by runtime peer A to the PIC of runtime peer B. The entries are not read or taken by runtime peer B. Thus, only write operations need to be access-controlled in the developed security model.

In the PeerSpace.NET entries can only be sent between runtime peers with the aid of the DEST property. As the transmission of entries between runtime peers is anyway access-controlled by the security model, the DEST property does not need to be considered concerning the security. The concept of the FlowId also does not need to be considered as it is not implemented in the PeerSpace.NET.

The requirements and the differences to the Secure Peer Model are clear now. Let us turn our attention to the design of the security model for the PeerSpace.NET. This chapter is divided into an architecture section and a part which addresses the operating principle of the rules which form the centerpiece of the access control. It concludes with the validation of the design concerning the requirements.

5.1 Security Architecture

The entire architecture for the security model is composed of an inter- and an intra-peer architecture and will be explained in this structure.

5.1.1 Inter-Peer Security Architecture

In the Peer Model, peers send entries to each other as described in section 3.1. They use an I/O component for the remote transfer of entries as depicted in Figure 5.1.

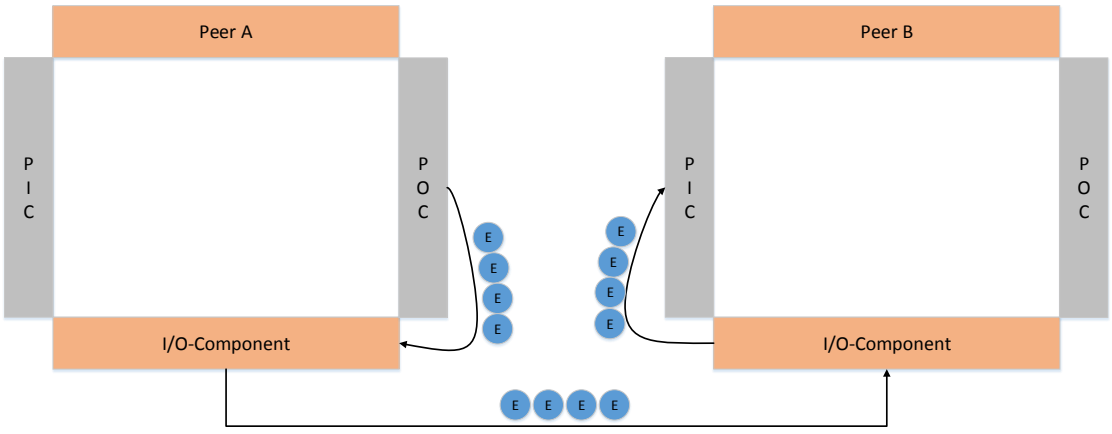

Figure 5.1: Transfer of entries between peers

The participants in the developed security model are the runtime peers, the users and the identity provider. When a user starts a security-enabled runtime peer, he/she must provide user id and the corresponding private key. The identity provider stores sets of user id, the corresponding public key and the user's security attributes. The identity provider's task is to validate whether a message was sent from the stated user and provide his/her security attributes in order to authenticate the message. The administration of users and their security attributes is centrally managed at the identity provider by one administrator. A general explanation about a public key infrastructure (PKI) can be found in section 2.1.5.

In order to prove the identity of the sending user and to verify that the entries have not been tampered during transfer, the entries are signed by the sender's private key and his/her id is also sent. The receiving peer contacts an identity provider which verifies the correctness of the obtained data and answers with the sender's security attributes if everything is alright. Then the receiving peer links these security attributes to the obtained entries, thus they get authenticated. Entries which cannot be authenticated are dropped. Before the entries are written to the peer's PIC, they are intercepted by the Access Manager, which decides according to the peer's policy whether the received entries are granted or denied to access the peer's container. The access is granted either for all entries or for none. Figure 5.2 illustrates the inter-peer architecture.

In order to ensure the intended receiving peer is not spoofed and provide confidentiality, peers transmit entries among themselves also by means of a TLS encryption, established with the aid of a certificate issued by a trusted authority. In this way computer-to-computer identity and confidentiality are ensured. Although the user's security attributes could be provided by the user's certificate, an identity provider is useful due to the following reasons anyway.

With the aid of the identity provider different users can run their runtime peers on the same computer and possess their own user accounts. If the security attributes were stored in a certificate which is valid for the entire computer, different users with different user accounts on one computer would not be feasible.

Further, changes of a user's security attributes can be managed centrally at the identity provider with no need of the user's cooperation. If the security attributes were stored in a certificate the user would have to install the new certificate holding the changed security attributes. Besides, the expired certificate would have to be specified on a revocation list.

The runtime peers are also connected to the identity provider via TLS in order to ensure the identity provider is not spoofed. When runtime peer B receives entries from runtime peer A, runtime peer B sends a request for the security attributes of runtime peer A's user. With the request runtime peer B sends the hash of the received entries, runtime peer A's signature and id. Thus the identity provider can validate the identity of runtime peer A by means of the received data. A more detailed description can be found in section 5.1.3.

5.1.2 Intra-Peer Security Architecture

After the inter-peer architecture has been depicted to get a rough overview of the functioning of the introduced security model, the security structure inside a peer is discussed next.

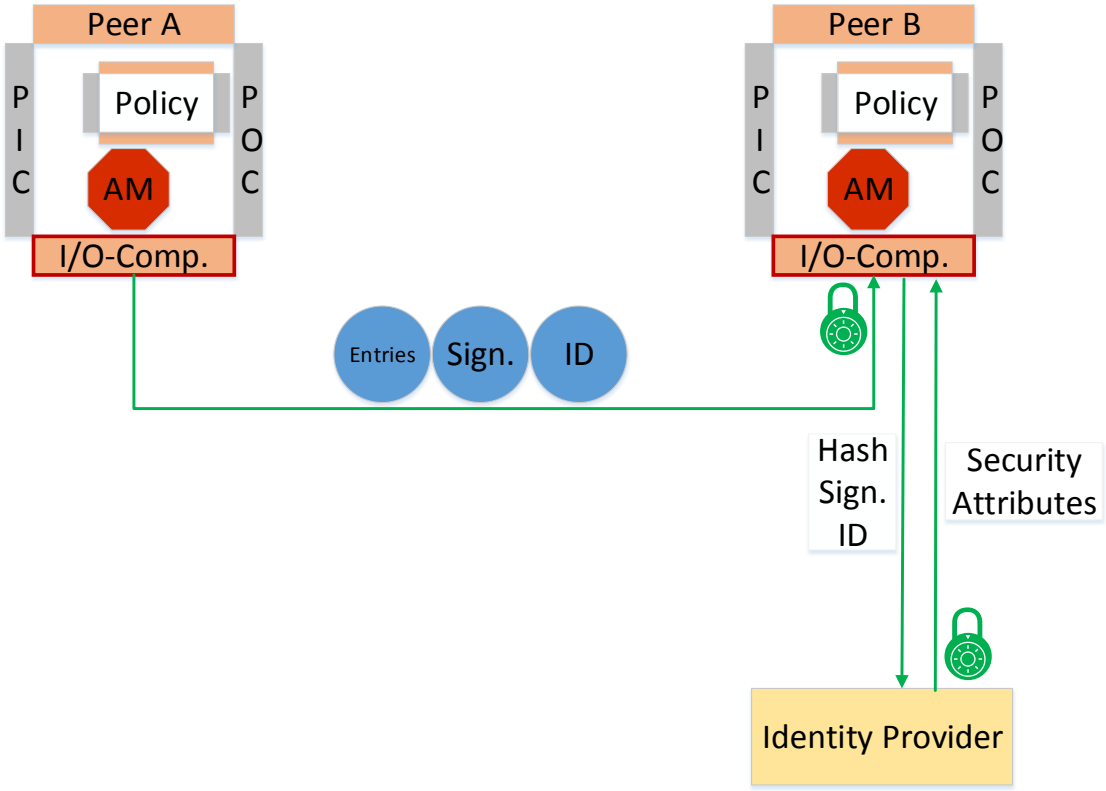

Figure 5.2: Transfer of entries between secured peers

Security Architecture in Sending Peer

Every peer holds security-relevant objects, which are among others the peer user private key and id. Before entries are sent to another peer, they are packed into a message which is hashed with a secure hash algorithm and then signed with the user's private key. This procedure happens in the peer's I/O component. The outgoing message contains the message holding the entries that are intended to be sent, the entry which holds the signature and an entry containing the user id. Figure 5.3 depicts this set of facts.

Security Architecture in Receiving Peer

Beside the user's private key and id a peer also holds an access manager, a policy peer and a TLS connection to an identity provider. These items are used for authentication and authorization of the received entries. First the receiving peer calculates the secure hash of the inner message (without the id and the signature entries). Afterwards it sends this hash, the received signature and the id to the identity provider which verifies the signature by means of the hash and the id. This assures that the entries have not been tampered during transfer and the sender possesses the claimed id. When signature, hash and id fit together, the identity provider sends back the sender's security attributes. These are attached to the received entries and thus the entries get

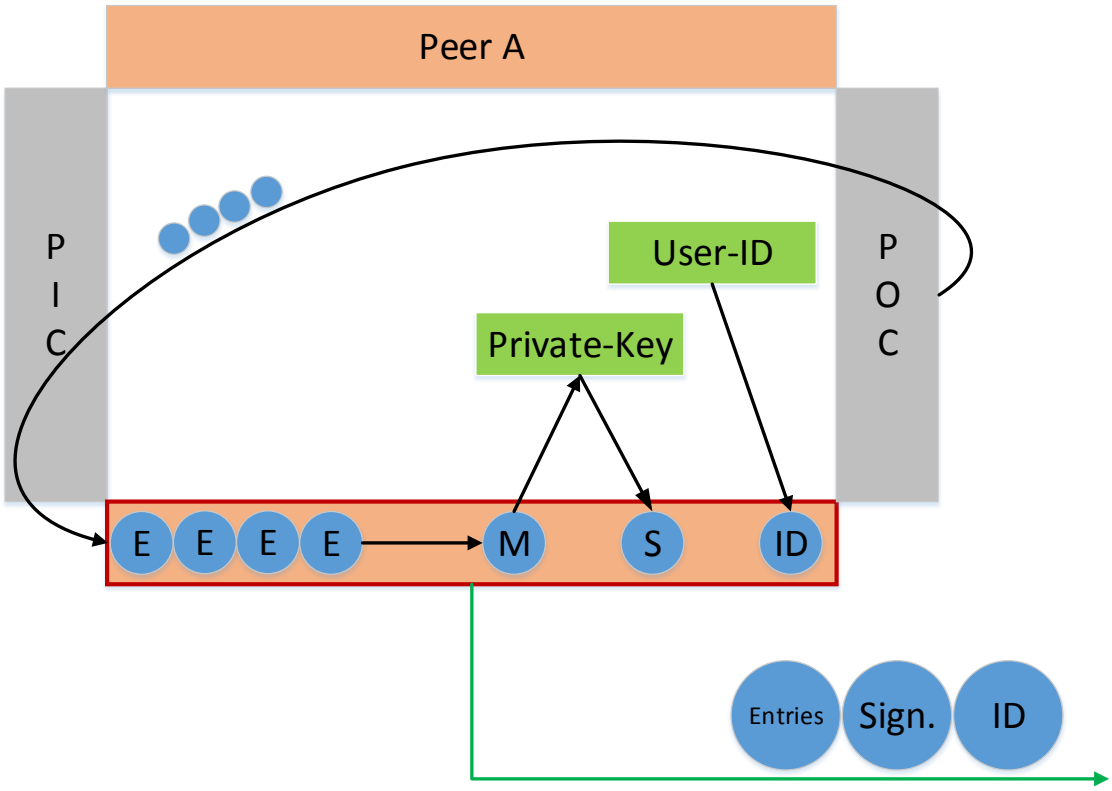

Figure 5.3: Security mechanism at sender side

authenticated. After the entries are authenticated they are handed to the access manager, which evaluates a security decision. It grants or denies the whole write operation, i.e. all entries are written to the peer's container or none. In order to make the security decision the access manager uses the security attributes attached to the entries, rules gathered from the policy peer, additional context information obtained from arbitrary (sub-) peers, the targeted container and the entries themselves. Figure 5.4 illustrates the whole procedure for receiving entries.

As peers can send and receive entries, a peer possesses both mechanisms: the one for sending and the one for receiving entries.

5.1.3 Security Components

As the architecture has been explained, we can focus on its components now.

Identity Provider

The whole access control is built upon the security attributes offered by the identity provider. Thus it must be ensured that the identity provider is not spoofed and its sent data has not been tampered. For this reason the identity provider holds a certificate issued from a trusted authority by which a TLS connection is established. The functioning of the identity provider is as follows:

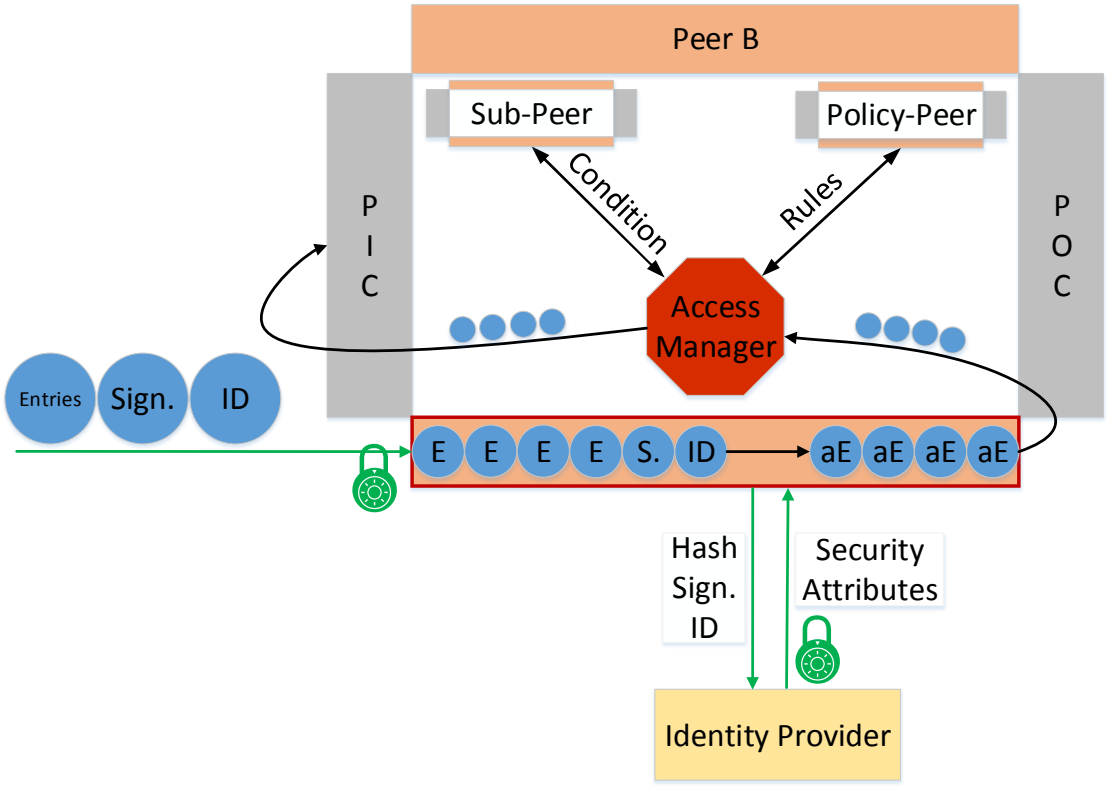

Figure 5.4: Security mechanism at receiver side

A peer sends the hash of its received entries along with the signature and id obtained from the sender, as mentioned before. The identity provider looks up the sender's public key by means of the received id. Therewith it verifies the signature against the hash, which represents the transmitted entries. If all data is correct, it is ensured that the sender is the one who he/she claims to be and the entries have not been tampered during transfer. When this is the case, the identity provider sends back the security attributes linked to the sender's id. Thereby the traffic to the identity provider is kept small, since only the hash is transmitted instead of all entries. Sending the hash also retains privacy.

Brute force attacks with the aim to gain security attributes and/or user ids from the identity provider are hampered as security attributes are not only queried by means of the user id. The security attributes and users are centrally managed by an administrator who has access to the identity provider.

Access Manager

The Access Manager is responsible to intercept entries, make an access decision and forward them to the desired target, when granted. To be more precise: the access manager intercepts entries that are intended to be written to a (sub-) peer's container, namely PIC or POC. In order to make the security decision, the access manager uses the security attributes attached to the

entries and the entries themselves. Further it needs the rules obtained from the policy peer and environmental context information gathered from arbitrary (sub-) peers. That way the access manager has direct access to all containers of the (sub-) peers. The targeted container is also needed in order to make a security decision. An operation is either granted or denied as a whole. When the access is denied, the respective entries are dropped and the action is logged. When the operation is granted the entries are written to the desired container.

As the access manager intercepts the entries and makes a security decision, it represents the Policy Enforcement Point (PEP) and Policy Decision Point (PDP) in one.

Policy Peer

The policy peer holds all the rules that build the security policy affecting the surrounding peer and its sub-peers. As the policy peer is a sub-peer too, it rules the access to itself with the same mechanisms used to rule the actual (sub-) peers, i.e. the access control bootstraps itself. Rule entries and remove rule entries, discussed in section 5.2, get from the parent peer's PIC to the policy peer via the security wiring. This wiring takes all rule and remove rule entries from a runtime peer's PIC and writes them to the policy peer's PIC. In this manner rules can be added and removed locally as well as remotely, if sufficient permissions are given.

As the rules are administrated in and retrieved from the policy peer, it represents the Policy Administration Point (PAP) and the Policy Retrieval Point (PRP).

5.2 Rules

Rules specify which entries are permitted to be written to specific containers under particular circumstances and build thereby the centerpiece of the access control model. They are realized as special entries in order to add them to a peer's policy peer during runtime possibly from remote, as long as sufficient permissions are granted. As most use cases can be modeled without deny rules, this access control model only uses permit rules in order to provide a good readability, comprehensibility and maintainability of the security policies and keep the evaluation algorithm as plain as possible. A write operation is only permitted if it is granted for every single involved entry, i.e. the operation is granted or denied as a whole. Thereby more than one rule could be necessary to grant an operation. That is the case, if one rule can only grant the operation for a part of the involved entries, and another rule can grant it for the remaining entries. None of the rules could permit the whole operation, but together they do.

5.2.1 Structure of Rules

A Rule is constituted of the following parts:

- **Id:** clear identification
- **Subject Property Template:** direct and indirect senders via security attributes
- **Resources:** peers and containers

- **Operation:** write (always)
- **Condition:** context condition (optional)
- **Scope Field:** content restrictions (optional)
- **Effect:** permit (always)

The purpose of the *Id* is to uniquely identify the rule, which is necessary when it is intended to get removed. The *Subject Property Template* defines the direct and indirect senders for which the rule is valid. They are identified by means of security attributes, as described in section 5.2.2. The *Resource* specifies for which (sub-) peers and containers targeted by the operation the rule applies. The *Operation* indicates which kind of access is ruled. As this access control model only needs to rule write operations, as described in the beginning of this chapter, the value of *Operation* is always *Write*. Due to readability, comprehension and expandability of the developed security model, it is specified nevertheless. The same applies to the *Effect*, which is always *permit*. The environmental context under which the respective rule is valid can be queried by the *Condition*. The *scope field* restricts the entries for which the rule is valid. A rule is applicable for a write operation, when the targeted container matches, the Subject Property Template is completely covered, as described in section 5.2.2, the particular entry type and the entry attributes fulfill the scope field and the condition evaluates to true. As several of the listed parts need a more accurate description, they are explained separately below.

5.2.2 Subject Property Template

The subject property template is the matching part to the subject property chain which in turn is derived from the delegation chain but defined with security attributes. Let us consider the following example: Runtime peer A sends an entry to runtime peer B which forwards the entry to runtime peer C which finally forwards the entry to runtime peer D. So the delegation chain is (*C* **for** *B* **for** *A*).

Assume that the user of runtime peer A is associated with the security attributes a1 and a2, the user of runtime peer B is associated with the security attributes b1 and b2 and the user of runtime peer C is associated with the security attributes c1 and c2. Note that a security attribute is a key:value pair. The subject property chain (*[c1:c1val, c2:c2val]* **for** *[b1:b1val, b2:b2val]* **for** *[a1:a1val, a2:a2val]*) corresponds to the delegation chain (*C* **for** *B* **for** *A*). The subject property chain is stored in the entry's coordination data and created when the entry is forwarded from a peer to another, as described later in this section. The security attributes are obtained from the identity provider which stores the user and the associated security attributes.

The subject property template is specified in the rule. A rule is applicable due to the subject property template when each element of the template is fully covered by the corresponding element of an entry's subject property chain. Thus a rule with the subject property template (*[c1:c1val]* **for** *[b1:b1val]* **for** *[a1:a1val, a2:a2val]*) is applicable for an entry with the subject property chain (*[c1:c1val, c2:c2val]* **for** *[b1:b1val, b2:b2val]* **for** *[a1:a1val, a2:a2val]*). Note that all security attributes of the first element of the subject property template must be a subset of the security attributes of the first element of the subject property chain. The same applies

to every element of the subject property template. Security attributes match when their keys and their values match. When only one security attribute of one element of the subject property template does not match the corresponding security attribute of the corresponding element of the subject property chain, the rule is not applicable for the respective entry.

Demand for the Subject Property Template

The subject property template is needed to specify rules for entries which are sent on behalf of other peers or users, respectively. This could also be achieved by impersonating the original sender. However, as no mutual trust can be assumed, this approach is not suitable.

Beside this, the subject property template is required to define rules for receiving entries that have been sent out by the own peer. Assume user Y is associated with the security attribute y1:y1val and sends an entry to a peer belonging to user S who is associated with the security attribute s1:s1val. Then a rule with the subject property template (*[s1:s1val]* **for** *[y1:y1val]*) at user Y's peer is applicable to the entry which was originally emitted by user Y's peer and is sent back by user S' peer.

Genesis of the Subject Property Chain

The subject property chain is created when entries are forwarded from a peer to another. Thereby the security attributes of the last sender are added to the subject property chain as first element. Let's consider the genesis of a subject property chain on an example illustrated in Figure 5.5.

Peer A sends entries to Peer B, which attaches Peer A's security attributes to the entries, specifically as first element in the entries' subject property chain. Therefore, Peer B queries the identity provider, as described in section 5.1.3. When the same entries are forwarded to Peer C, it inserts Peer B's security attributes as first element in the subject property chain and the security attributes from Peer A are moved to the second position. Finally when the entries have arrived at Peer D, their subject property chain is as follows:

Subject Property Chain [0] = c1:c1val, c2:c2val

Subject Property Chain [1] = b1:b1val, b2:b2val

Subject Property Chain [2] = a1:a1val, a2:a2val

The order of the security attribute chain is reverse to the sending order, whereby the direct sender is always on the first position. Note that all users in the subject property template are implicitly trusted, as their peers could manipulate the elements of the subject property chain. This manipulation would not be recognized as only the last sender is authenticated every time entries are received. Thus it is not sufficient to state only the original sender of entries, rather all trusted participants from the delegation chain should be specified.

Wildcard Support in the Subject Property Template

For the case that it does not matter who is the sender on a certain position in the subject property chain, wildcards are supported in the subject property template. For Peer D from the previous example the following subject property template would also be satisfied:

Subject Property Chain [0] = c1:c1val, c2:c2val

Subject Property Chain [1] = * (wildcard for one element)

Subject Property Chain [2] = a1:a1val, a2:a2val

As an element of the subject property template must be a subset of the sender's security attributes in order to be satisfied, a wildcard for one element of the subject property template is realized by stating no security attributes in the respective element.

When an arbitrary number of senders from the delegation chain are irrelevant, e.g. only the original sender is relevant of access control, another wildcard can be employed. Considering the same example again, the following subject property template applies for Peer D:

Subject Property Chain [0] = ** (wildcard for an arbitrary number of elements)

Subject Property Chain [1] = a1:a1val, a2:a2val

Subject Property Chain for Locally Written Entries

As entries that cannot be authenticated, i.e. that have no subject property chain, are dropped or rather no rules with a specified subject property template will grant access to them, a solution for entries that are written locally to a peer has to be found. "Locally written" means that the entries are not obtained from another peer, rather the peer owner writes them directly to the peer.

Locally written entries will obtain a subject property chain with an element which grants these entries access to the local peer and its sub-peers. This access is implicitly granted and does not have to be defined in rules. When these entries are sent to another peer, the element which grants local access is removed from the subject property chain. These entries are authenticated at the receiving peer, using the identity of the local runtime user, whereby a new element of the subject property chain is added.

Subject Property Chain of Entries Emitted by Services

Services may not and cannot make changes to the subject property chain of entries. When a service emits an entry, it is necessary to assign the entry a subject property chain due to the same reason depicted in the previous paragraph. It can be chosen whether the emitted entry obtains the same subject property chain as if the entry was locally written to the peer or the emitted entry's subject property chain is derived from the entry that was the first argument of the service. This means for future access control that the entry is either emitted by the peer itself or on behalf of the subjects from the first argument of the service.

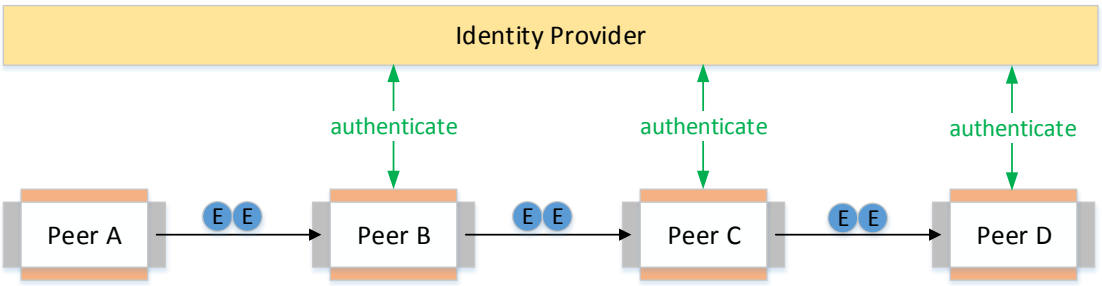

Figure 5.5: Delegation chain in the peer space

5.2.3 Condition

The environmental context under which a rule applies can be modeled with arbitrary entries in any local containers of (sub-) peers. With the aid of the optional condition this context is evaluated by means of the conjunction of condition predicates. These are specific queries against defined containers, which yield *true* when they are satisfiable and otherwise *false*. Condition predicates are bootstrapped with the query mechanism provided by the PeerSpace.NET for the action and guard links. Such a query is constituted of an entry type, an additional predicate, a count with a relation operator and the operation, which can be *read, write* and *take*. As the condition (predicate) may not change anything, its query operation is always *read*. A condition predicate also specifies the targeted container and its boolean outcome may be negated.

Assume a peer possesses sub-peer A and sub-peer B. For the peer a rule can be defined which is, e.g., only applicable if sub-peer A's POC contains at least two entries of type *string* and no entries of type *int* or sub-peer B's PIC contains at most three entries of type *bool*.

A condition predicate which shall evaluate to true when no entries of a certain type are present can be realized by defining a negated condition predicate that queries one entry of the particular type. Thus when one or more entries of the particular type are present, the negated condition predicate evaluates to false. Otherwise, i.e. when no entries of this type are available, the negated condition predicate evaluates to true. The same principle can be used to specify a condition predicate that evaluates true when e.g. at most three certain entries are present.

A condition predicate's query may also use dynamic values which are received from the entry whose access control decision is evaluated. Thus, e.g. a rule can be specified which grants entry A access, only when a registration entry B is available that possesses the same id like entry A.

5.2.4 Scope Field

With the aid of the optional scope field a rule can be restricted to be valid only for entries of specific types and with certain attributes. Static scope fields specify with the aid of static values whether the rule is applicable to an entry or not. A static scope field may define: *id = 123* e.g. The according rule is applicable for entries with the id of "123". Dynamic scope fields define dynamic values obtained from the entry's subject property chain to restrict the rule for certain entries. For example a dynamic scope field may define *id = $id*. The according rule is

applicable for all entries where the id matches the id of the sender. Thus rules can be stated that are applicable for entries that were sent by their owner. A more detailed description can be found in section 6.3.1.

5.2.5 Remove Rules

Rules are realized as special entries and thereby distributed via the PeerSpace.NET transport mechanism, i.e. its wirings. As wirings are located at sender side and cannot be added to peers from remote, as discussed in section 3.2, it is not possible to remove rules via a dynamic take wiring from outside the peer. This approach would be preferable because it is bootstrapped by the PeerSpace.NET. However, another mechanism for removing rules will be introduced. This mechanism uses Remove Rules which are special entries like rules and likewise distributed. When a remove rule is sent to a peer and forwarded to the peer's policy peer, the remove rule causes the elimination of the respective rule identified via the id. Sufficient privileges must be given to the sender in order to remove a rule. The access manager has direct access to all containers of the (sub-) peers and thus to the policy peer's PIC, where the rules are located. Thus the access manager executes the deletion of rules. It would also be possible to delete rules with the aid of dynamic wirings. However, an own dynamic wiring would have to be created and deleted for each removal of a rule.

5.3 Rule Evaluation

An operation is either granted or denied as a whole. Thus access must be granted to every entry involved in an operation to grant that operation. Access to a group of entries cannot be granted at once, because the dynamic scope field and the dynamic condition of a rule must be evaluated for every single entry. To avoid that n*m rules are evaluated for n rules and m entries, all rules which are not applicable due to the peer and the container are dropped first. Then the remaining rules are ordered concerning the probability that they are applicable. For each entry the rules are evaluated in this order and as soon as one rule grants access to the entry, the remaining rules are skipped for that entry and the rule evaluation continues with the next entry. A more detailed description of this mechanism can be found in section 6.3.2.

5.4 Parallels to XACML

The access manager represents the Policy Enforcement Point (PEP) and the Policy Decision Point (PDP) like mentioned in section 6.3.2. The policy peer represents the Policy Administration Point (PAP) and the Policy Retrieval Point (PRP), as already noted in section 5.1.3. As a rule's condition queries the (sub-) peers' containers, they represent the Policy Information Point (PIP).

5.5 Fulfilled Requirements

In order to show that the design of the security model is suitable, the previously set up requirements from section 4.1 are verified for fulfillment.

Usage of attribute-based access control (ABAC)
Security attributes are attached to the transmitted entries with the aid of the identity provider. Rules are selected to be employed by means of these security attributes (beside other things).

Trusted user administration
The identity provider, whose identity is proven by means of a certificate issued by a trusted authority, is the entity where users and their security attributes are administrated.

Access-controlled resources
The access to the containers within the Peer Space, i.e. the PICs and POCs, is controlled. The containers can be accessed through read, write and take operations, but due to the characteristics of the PeerSpace.NET, discussed in section 3.2, only write operations are ruled.

Expressive rules
A rule can specify the entry's type and additional attributes by means of the scope field, described in section 5.2.4, and may only be valid under a certain environmental context, queried by the condition, discussed in section 5.2.3.

Wildcard support
Wildcards are supported for the scope field, the condition and the Subject Property Template in order to facilitate the creation of general rules.

Permit rules only
The designed access control model uses permit rules only.

Support for delegation
With aid of the subject property template rules can be defined for delegated entries by means of all of their senders.

Every peer possesses its own security policy
Every peer holds its own policy peer which in turn holds the rules valid for the particular peer and its sub-peers.

Management of rules is bootstrapped
As the policy peer is a sub-peer and the rules are realized as special entries, the management of rules is bootstrapped with the Peer Space itself.

The security policy can be changed during run time
This requirement is satisfied, as the policy peer is a sub-peer and rules and remove rules are implemented as special entries, which are sent during run time.

Setting the security policy remotely
As rule and remove rule entries are transmitted like conventional entries, i.e. remotely, this issue is also fulfilled.

Enforcement of access control is optional
Within a peer's configuration, which acts as a blue print in order to produce multiple peer instances, access control can be enabled or disabled. By default it is enabled. When access control is enabled for a runtime peer, it is automatically activated also for its sub-peers.

Integrity and authenticity
This is achieved through a public key infrastructure, whereby the sending peer uses its private key to sign the entries and the identity provider verifies the signature with aid of the respective public key.

Confidentiality
Confidentiality is given though the deployment of an TLS encrypted connection between peers, as well as between peers and the identity provider.

Scalability
Every runtime peer holds its own policy peer and every user can administrate its own rules or delegate this task. Attribute-based access control is used in order to decouple rules from users, which is convenient when new users are added. Further the traffic to the identity provider is kept small by only transmitting the secure hash instead of the whole message, in order to a verify the signature.

Usability
To employ the introduced access control model, the TLS connections must be established and rules must be created. Besides, the acting users must be registered at the identity provider. All these procedures are not to complicated and thereby practicable. Finally the employment of permit rules only increases the comprehensibility.

Maintainability
When new users are added, rules need not necessarily be adapted, since attribute-based access control is deployed. Every user can maintain the policy of his/her own peer, but this task can also be delegated to a certain user.

Performance
The performance impact caused by the introduced security model will be evaluated by means of a benchmark test (see Section 7.2).

CHAPTER 6

Implementation

The design of the security model is now established and theoretically validated, so it can be implemented.

Therefore we analyze the implementation of the existing *Application Peer* and explain its required security adaptions. Next the implementation of the necessary authentication mechanisms are examined, followed by an explanation about the realized authorization with its required elements, like the access manager, the policy peer and rules.

6.1 Application Peer

As the security mechanisms reside in the *Application Peers*, they are studied next.

6.1.1 The Functioning of the Application Peer in the PeerSpace.NET

Within the PeerSpace.NET there are currently only *Application Peers*, which represents the actual peers. When an application peer's run method is executed, the peer is converted to a *Runtime Peer*. Thereby it obtains an *XcoAppSpace* for remote communication with other runtime peers and the sub-peers, which are specified in a list, are annexed.

Sub-peers are also application peers, but in contrast to the runtime peer, the run method has not been executed. An application peer can either be run or annexed to a runtime peer, which converts it to a sub-peer. So sub-peers cannot be run and thus no nested sub-peers are possible.

An application peer has methods in order to write entries locally into its PIC or POC. These methods are either called by the user, by a local wiring's action, or by the communication layer when entries have been received from remote.

However, all these methods converge to the *EmitRespectingTimeProperties* method, which basically writes the entries to the desired container taking into consideration the entries' TTS and TTL properties. Subsequently the corresponding wirings are run, if possible.

When a wiring is run, it reads or takes specified entries from a source container and writes them into an wiring-intern container called Entry Collection. Afterwards, services which are

specified in the wiring are called. These services may use entries from the entry collection as arguments and may emit newly generated entries to the entry collection. Next the wiring reads or takes specified entries from the entry collection and writes them to a defined target, which is a (sub-) peer's container.

Figure 6.1 depicts the architecture of runtime peers and sub-peers and the transfer of entries.

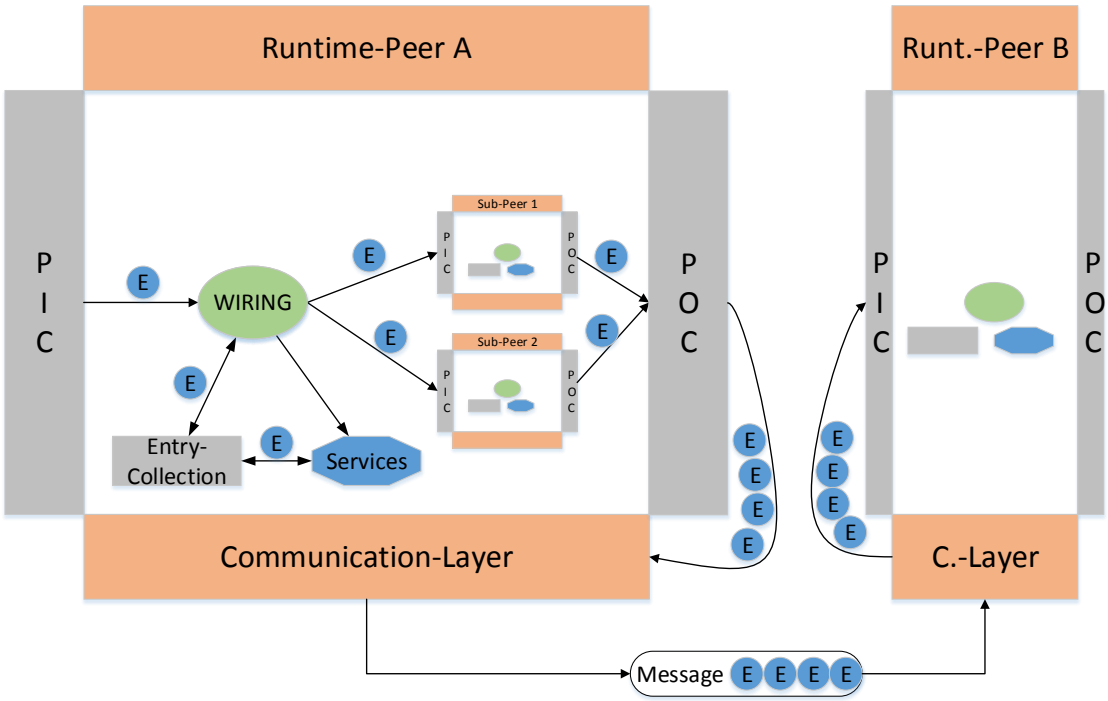

Figure 6.1: Functioning of the PeerSpace.NET

6.1.2 Security Adaptions in the Application Peer

Application peers are instantiated by means of a peer configuration, which acts as a blue print. This peer configuration also specifies whether the particular peer will be secured or not when it is converted to a runtime peer. When a runtime peer is secured, its sub-peers are automatically also secured, no matter what is defined in their peer configuration.

A secured runtime peer needs the user's id and private key in order to properly communicate with other secured runtime peers. Therefore a file path is stated in the peer configuration which locates a configuration file containing the user's id as string and his/her private key as XML string, which is a convenient way to import the cryptographic key.

As an entire runtime peer is owned by one user, the access control model applies when entries are transferred between runtime peers. The mechanism for authentication which enables authorization has to be integrated in the procedures of sending and receiving entries.

Since the id and key are only required for secured runtime peers, they are first imported from the configuration file when the peer's run method is executed. Thereafter they are passed to the

peer's communication layer, which needs the information in order to be able to send and retrieve entries with authentication support.

6.2 Authentication

The explanation starts with the implementation of authentication support at sender side. Before the security mechanism for sending entries can be implemented, the procedure that occurs when entries are sent has to be studied.

6.2.1 Preexisting Sending Mechanism

In order to get an idea of what happens in the PeerSpace.NET when entries are sent to remote runtime peers, the mechanism is examined by means of the flowchart in Figure 6.2 first.

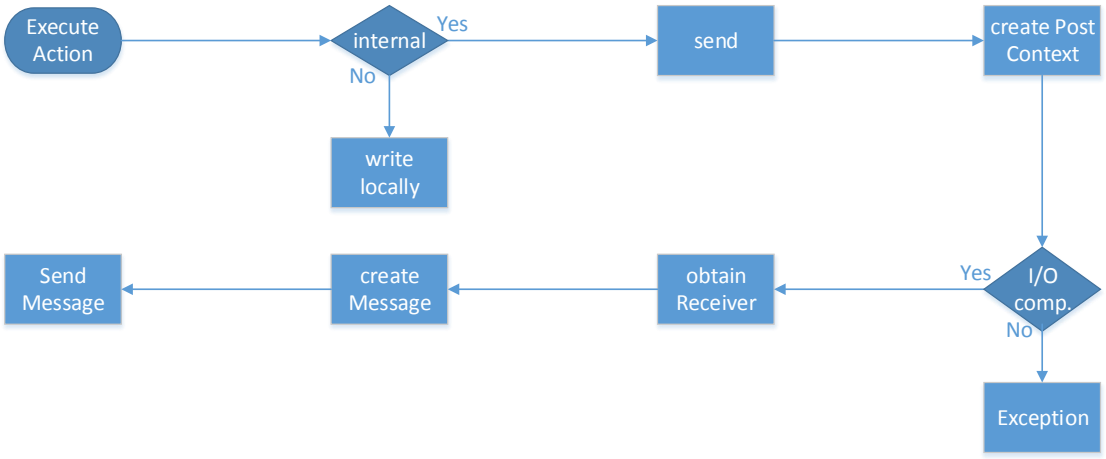

Figure 6.2: Sending procedure in the PeerSpace.NET

When a wiring is run and it executes its *action*, entries are sent either to a local or remote destination. Thus *Execute Action* forms the starting point for this flowchart. It is evaluated whether the wiring's destination is internal or not. If it is internal, i.e. PIC, POC or a sub-peer, the entries are written to the particular container.

When the destination is external, the entries are going to be sent over the peer's I/O component. Therefore a *PostContext* is created containing the entries, the sending peer and the peer address of the destination.

The next step is that it is verified that the sending peer is no sub-peer and possesses an I/O component. If the sending peer is no sub-peer and possesses an I/O component, an URI to the receiver is obtained with the aid of the destination's peer address from the *PostContext*. Next a *Message* containing the entries from the *PostContext* is created and finally transmitted to the obtained receiver.

With the aid of the UML diagram in Figure 6.3, which depicts the implementation of the peer (*Application Peer*) and the I/O component (*XcoAppSpaceCommunicationLayer*), a more

detailed description of the involved methods follows. The methods from the UML diagram are part of the PeerSpace.NET implementation and already exist. The developed security model is integrated into some of these methods.

Figure 6.3: UML diagram of the application peer

For the following explanation it is assumed that the destination is remote and that the entries are going to be sent via the I/O component.

There are two *SendTo* methods, which are basically the same. The only difference is that one of these takes one entry as argument and the other a collection of entries. Further they demand the peer address of the destination. The two callbacks are not discussed since they have no relation to the security.

The *SentTo* method creates a *PostContext* which includes the destination peer address, the entries and the address of the own peer, which acts as sender. Then it calls the *Post* method with the *PostContext* as argument, which checks that the sending peer is no sub-peer and possesses a communication layer.

Then the *PostContext* is handed to the *Post* method of the communication layer which is an instance of *XcoAppSpaceCommunicationLayer*. With the aid of the destination peer address stated in the *PostContext*, the *ObtainClientForAddress* creates a stub for the *XcoAppSpace* of the receiver.

Next the *CreateAndPrepareMessage* method wraps the entries from the *PostContext* in a *Message*, which uses the *XcoAppSpace* for transportation. Finally the *Message* is sent to the receiving *XcoAppSpace*.

6.2.2 Implementation of Sending with Authentication Support

In order to enable authentication on receiver side, the user's signature and id are added to the *Message*. The id and signature are each wrapped in an entry. The actual entries which are intended to be sent are packed into a *Message*, called net message, which in turn is wrapped in an entry. So the final message always contains three entries in the first tier, namely the net message, the id and the signature.

The examined approach has already been visualized in Figure 5.3 and is explained with the aid of the flowchart in Figure 6.4.

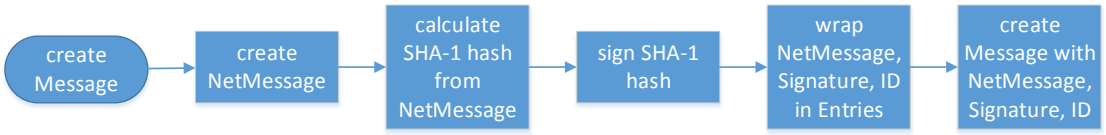

Figure 6.4: Creating a message that can be authenticated

At sender side authentication is supported by the message, which has to be adapted for that reason. So the place where the message is created is the convenient location for the implementation of authentication support at sender side. The message is created in the *CreateAndPrepareMessage* method, which resides in a runtime peer's I/O component, i.e. the *XcoAppSpaceCommunicationLayer*, as depicted in Figure 6.3. The implementation of the explained security mechanism for sending is established by only changing this one method.

In the changed *CreateAndPrepareMessage* method the signature is created by the *SignObject* method from the static *SignUtility* class. It gets the net message and the user's private key as arguments and calculates the secure hash (using SHA-1 [60]) from the net message. Then

an instance of the *RSAPKCS1SignatureFormatter* is created with the private key as parameter. This class is shipped with .NET. On the instance of the *RSAPKCS1SignatureFormatter* the *CreateSignature* method is called with the previously calculated SHA-1 hash from the net message as parameter. The *CreateSignature* method returns a signature which in turn is returned by the *SignObject* method.

The net message, the signature and the id are wrapped into entries by the *WrapObjectIfNecessaryAndSetProperties* method from the *EntryUtilities* class, which have already existed within the PeerSpace.NET.

6.2.3 Preexisting Receiving Mechanism

When entries are received at a peer's *XcoAppSpace*, the *ProcessMessage* method, which resides in the *XcoAppSpaceCommunicationLayer*, is triggered. This method forwards the entries to the peer's *Emit* method, which in turn writes the entries to the PIC and tries to run the corresponding wirings. The procedure of receiving is illustrated in the flowchart in Figure 6.5. Note that the description and diagram only illustrate the relevant steps and make no claim to completeness.

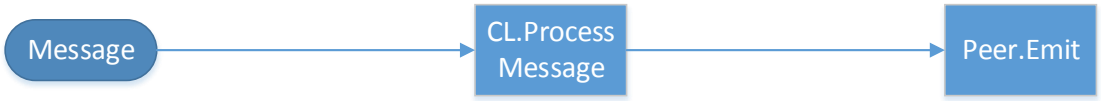

Figure 6.5: Receiving mechanism in the PeerSpace.NET

6.2.4 Implementation of Receiving with Authentication

When a message is received, the three contained entries, namely the net message, the signature and the id are unwrapped and verified. If any of them is invalid, the message is dropped.

When all received data is correct, the secure hash of the net message is calculated and sent along with the id and the signature to the identity provider. The identity provider in turn verifies by means of the received data the entries and the original sender and returns his/her security attributes, if all data is correct. If not, it returns null with the result that the action is logged and the entries are dropped. Otherwise the security attributes are attached to the received entries, which are thereby authenticated. The examined approach has already been visualized in Figure 5.4 and is sequentially depicted as a flowchart in Figure 6.6.

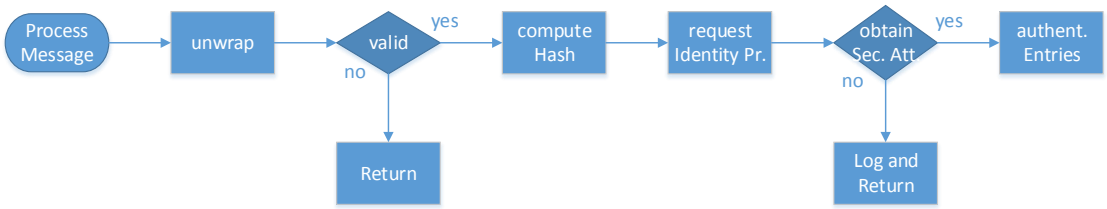

Figure 6.6: Security mechanism for receiving

This part of the security model takes place when entries are received, so the appropriate location for its implementation is where retrieved messages are processed. The flowchart in Figure 6.7 depicts the procedures for receiving entries and marks the procedure where the security takes place. This is at the method *ProcessMessage*, which resides in the *XcoAppSpaceCommunicationLayer*. The implementation for authentication of received entries is established by only changing this one method.

In the changed *ProcessMessage* method the id, the signature and the net message are unwrapped and the secure hash of the net message is calculated. The secure hash, the signature and the id are handed to the identity provider and therewith the user's security attributes are requested. This is established by the *GetSecurityAttributes* method from the static *ConnectToIdentityProvider* class.

When security attributes are returned by the identity provider and thus by the *GetSecurityAttributes* method, they are inserted into the entries' subject property chain. In this way the entries have been authenticated.

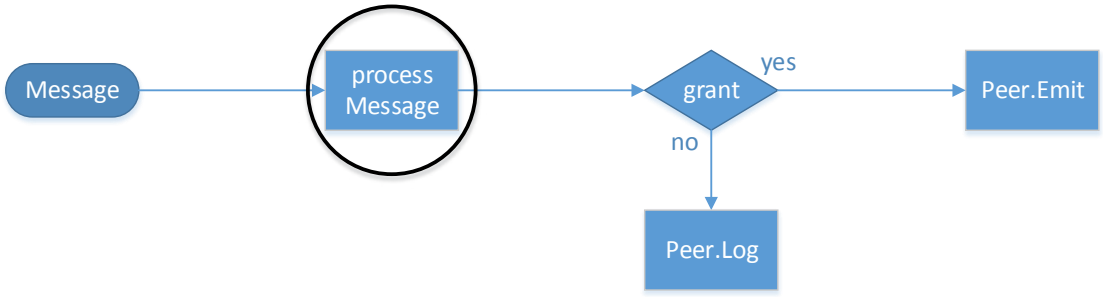

Figure 6.7: Integration point of the security mechanism

6.2.5 Subject Property Chain for Locally Written Entries

When a user writes entries locally to a peer with one of the *Emit* methods, they do not pass the *XcoAppSpaceCommunicationLayer*. Thus they are not authenticated and accordingly no access control can be performed on them. For this purpose the *SudoEmitToPic* and *SudoEmitToPoc* methods have been created.

Within these methods entries gain local access by adding the security attribute *LocalAdmin* in their subject property chain, before they are written to the corresponding container. Thus they are granted access to the whole runtime peer, but when they are sent to a remote peer, the security attribute *LocalAdmin* is deleted from the sending peer and the entries get authenticated at the receiving peer with the sender's security attributes.

6.2.6 Subject Property Chain of Entries Emitted by Services

Entries which have been emitted by a service do not pass the *XcoAppSpaceCommunicationLayer*, similar to entries locally written to a peer by the user. The entries have to get a subject property chain in order to conduct authorization decisions on them. In contrast to the locally

written entries, a subject property chain with security attributes exists, namely the one of the entries taken from the service as argument. The approach is to either attach a copy of the subject property chain from that entry which forms the first argument of the service to the emitted entries or to attach a subject property chain containing the security attribute *LocalAdmin*. The created *DerivedAuthenticationEmit* method enables a service to emit entries with a copy of the subject property chain from the entry which has been taken from the service as first argument. The created *SudoServiceEmit* method emits entries with a subject property chain containing the security attribute *LocalAdmin*.

6.2.7 Identity Provider

The Identity Provider's task is to verify the sender of a message and respond with the appropriate security attributes. Tampering of the entries during transfer would be recognized by checking the sender's signature. As the identity provider bootstraps the authentication and therewith the whole security model it is very important that it cannot be spoofed, because faked security attributes could be distributed. In order to ensure the identity provider is the one it claims to be, it is connected via TLS with the aid of a trusted certificate.

The Identity Provider is realized as WCF service, hosted in a stand-alone console application that holds the ids, public keys and security attributes of the registered users within a dictionary, whereby the user's id serves as key. The user registrations are read from a text file and copied to the dictionary during the startup of the identity provider. A local administrator can change users and their security attributes by modifying this file. A user interface for administrating this file can be later implemented to increase the usability.

It would be conceptionally preferable to bootstrap the identity provider with the Peer Space and realize it as runtime peer, but there are a couple of reasons why we decided to implement it as WCF service.

Due to the important role of the identity provider in the security model it is indispensable that request and responses are not lost during transfer, even if the connection is temporarily broken. With the employment of WCF services this feature can be ensured as WCF uses *Message Queuing (MSMQ)* which implements a reliable message protocol [45].

WCF supports also routing, wherewith load balancing can be established. This is useful when the Peer Space scales, as the identity provider forms a bottleneck of the security model.

The necessary TLS connection to the identity provider can be established using WCF services. If the identity provider was implemented as runtime peer, the TLS connection would depend on the Peer Space‘s I/O component, which would be a drawback when the communication layer is changed.

Besides, WCF hosted services are interoperable, thus the identity provider may also be deployed for other implementations of the Peer Model, assuming they use a similar security concept.

6.3 Authorization

In the context of this thesis, authorization defines whether certain entries are allowed to be written to a specific container or not. The decision depends on many factors like the security attributes of the direct and the indirect senders, the rules, the content of the entries and the context information.

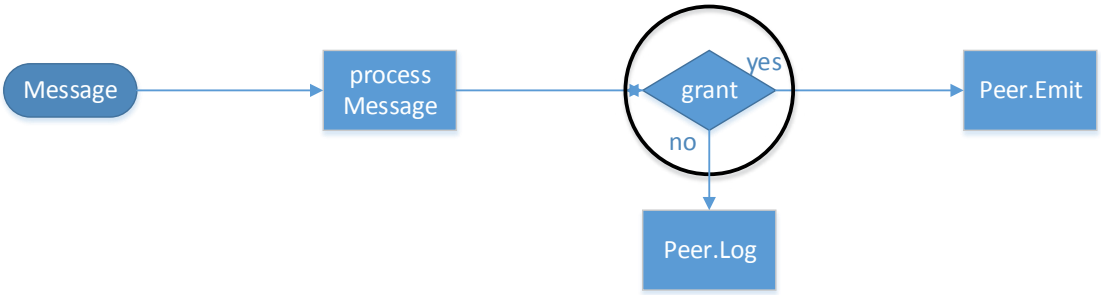

Figure 6.8: Point of interception for access control

6.3.1 Rules

In order to contain all required information, a rule consists of the following fields.

- Id
- Guarded Peer Containers
- Subject Property Chain
- Operations
- Scope Field
- Condition
- Rule Effect

The *Guarded Peer Containers* specify the peers and their containers for which the rule is valid by the name of the particular peer and container. The *Subject Property Chain* is a list of dictionaries which in turn hold the security attributes from the particular sender of the delegation chain. The *Operation* is always *write* and is stated for clarity and extensibility, like the *Rule Effect*, which is always *Permit*.

Scope Field

The scope field is constituted of a hash set of types and an additional predicate and specifies for which entries the rule is applicable. The types specify for which entry types the rule is valid and with the aid of the additional predicate more accurate specifications can be accomplished.

The additional predicate is implemented as delegation method which takes an entry as argument and returns a boolean. Thus an arbitrary boolean function may be created, which can access every part of the entry, i.e. the application data as well as the coordination data. Therewith static and dynamic selection criteria can be defined as shown in the following two examples.

When an additional predicate is defined, it is recommended to specify only a single entry type to prevent type conversation problems during the evaluation of the additional predicate. However, when no additional predicate is stated, the possibility to define more than one type in the scope field is useful.

Static Scope Field Example

The rule in this example shall be applicable for the *Student Entry* with the matriculation number *MNr* 0425266. So within the scope field the following delegation method, which is implemented as *Lambda Expression*, has to be stated.

```
entry => ((Student)entry.Data).MNr == 0425266;
```

Listing 6.1: Lambda expression for static scope field

With *entry.Date* the application data of the entry, i.e. the Student object, is accessed. In order to address the students matriculation number *MNr*, the application data has to be cast to *Student*.

Dynamic Scope Field Example

In the next example the rule shall apply to the *Student Entry* which belongs to the sender. The matching is verified by the matriculation number. Thus the matriculation number in the entry must be that of the sender. Thereby the *Lambda Expression* accesses the application data and the coordination data of the entry.

```
entry => ((Student)entry.Data).MNr ==
   entry.SubjectPropertyChain[0][SecurityAttributes.MNr];
```

Listing 6.2: Lambda expression for dynamic scope field

The left part of the boolean expression is the same as in the previous example, since it will be validated if the rule is applicable by means of the matriculation number. The difference to the former scope field is that the matriculation number is not compared to a static number, rather it is compared to the one of the sender. Thus the security attribute *MNr* of the last sender is used, which is obtained from the entry's subject property chain whereat the index *0* addresses the last sender.

Due to clarity the lambda expressions shown in the previous examples do not perform any checks for avoiding runtime exception, like *cast* or *key-not-found* exceptions.

As the query mechanism in the PeerSpace.NET consists of *type, additional predicate, count, relational operator* and *operation*, which constitutes a super set of the scope field, it could be bootstrapped therewith. However, as the scope field only uses *type* and *additional predicate*, this would be an overkill. Further, queries are performed against containers, whereby the scope field is applied directly to entries, when ruling write operations.

Condition

In contrast to the scope field the condition is bootstrapped with the PeerSpace.NET query mechanism. With the aid of the condition, the environmental context under which the rule applies can be specified. The environmental context is modeled with arbitrary entries in any container within the runtime peer. Thus the condition is the boolean result of a stated combination of defined queries against specific containers.

Such a single query, called condition predicate is constituted of a PeerSpace.NET query, the targeted peer and container name and an optional logical negation. The condition predicate has a boolean outcome, whose value depends on whether the query was satisfiable and if the result is negated. The query of the condition predicate possesses the following fields.

Operation = QueryOperation.Read

Type

Relation = QueryRelation.Exactly

Amount

AdditionalPredicate

The *operation* is always read as the condition must not make changes within the Peer Space. The *relation* is exactly, although a query can also be satisfied when more than the stated entries are available. So the relation could also be stated as more or exactly. If it is desired to create a condition that evaluates to *true* when less than a certain count of entries are located, this can be realized with the aid of the optional negation.

The *type, amount* and *additional predicate* relate to the queried entries and are specified by the rule's creator. The additional predicate is realized with a delegation method, which takes the queried entry as input and returns a boolean. In order to enable a dynamic condition, where the entry for which the rule is evaluated can be involved in the additional predicate, the additional predicate is overloaded with a delegation method that takes two entries as input and returns a boolean. One of these two entries is the entry for which the rule is evaluated and the other entry is the same as in the other delegation method.

The condition predicates are connected with logical connectors, namely conjunctions and disjunctions. The condition consists of a list of condition predicates and a list of logical connectors, whereby the former list possesses one element more then the latter.

The condition is sequentially evaluated as follows. The result of the first condition predicate is combined with the second one according to the first element of the logical connectors. Then

this result is combined with the outcome of the third condition predicate corresponding the second element of the logical connectors and so forth.

For example, the following condition evaluates to true when at least five entries of type string with the length of ten characters are located in sub-peer A's PIC and less than two entries of type int are located in the sub-peer B's POC. Note that the amount of the second condition predicate is two, but due to the enabled negation, this predicate evaluates to true when less than two entries are located.

Source = Sub-Peer A's PIC
Type = string
Amount = 5
AdditionalPredicate = length=10
Negation = false

Logical Connector = AND

Source = Sub-Peer B's POC
Type = int
Amount = 2
AdditionalPredicate = true
Negation = true

Figure 6.3 depicts a code example of an entire rule, which is valid for entries of the type string which contain the character "u" (scope field). The rule is valid for the entries that were sent from a supervisor on behalf of a student (subject property template). The condition for the rule is the same as in the previous condition example.

```
var rule = new Rule
                {
                    Id = "247",
                    SubjectPropertyTemplate = new
                       List<Dictionary<SecurityAttributes,
                       HashSet<string>>>
                    {
                        new Dictionary<SecurityAttributes,
                           HashSet<string>> { {
                           SecurityAttributes.Role, new
                           HashSet<string> { "Supervisor" } }
                           },
                        new Dictionary<SecurityAttributes,
                           HashSet<string>> { {
                           SecurityAttributes.Role, new
                           HashSet<string> { "Student" } } }
```

```
            },
            Scopefield = new Scopefield(new
               HashSet<Type> { typeof(string) },
               e=>e.Type == typeof(string) &&
               ((string)e.Data).Contains("u")),
            Condition = new Condition
            {
                ConditionPredicates = new
                   List<ConditionPredicate>
                {
                    new ConditionPredicate("SubPeerA",
                       "PIC", typeof(string), 5,
                       LogicalNegation.DontNegate,
                       e=>e.Type == typeof(string) &&
                       ((string)e.Data).Length == 10),
                    new ConditionPredicate("SubPeerB",
                       "POC", typeof(int), 2,
                       LogicalNegation.Negate, e=>true)
                },
                LogicalConnectors = new
                   List<LogicalConnector> {
                   LogicalConnector.And }
            }
        };
```

Listing 6.3: Example of a rule

The condition, scope field, guarded peer containers and the subject property template implement wildcards as follows. If the particular field is not set, i.e. it references null, the rule applies always concerning the respective variable. The wildcards for elements of the subject property template are specified with the enums *Wildcard.ForOneElement* and *Wildcard.ForArbitraryElements*.

6.3.2 Access Manager

The access control decision, whether authenticated entries are allowed to be written to a certain container or not, is evaluated by the access manager. While Figure 5.4 illustrates the interaction of an access manager in a runtime peer, Figure 6.8 shows where in the receiving chain the security decision resides.

Every peer has the field *AccessManager* of the type *IContainerHoldingAccessManager*, which offers the *GetPermission* method to evaluate whether entries are allowed to be written to a certain container or not. The type is an interface due to two reasons. Firstly, it enables extensibility, e.g. allowing the integration of an access manager which acts on access control lists. Secondly, when an application peer is instantiated and it is not set to be secured, an instance of

the *GrantAllAccessManager* will be created and assigned to the field *AccessManager*. In this way, every write operation is granted and no access control is performed.

There is only one instance of the actual access manager, i.e. the *AbacAccessManager*, in the whole runtime peer. Thus the *AbacAccessManager* is instantiated when the *Run* method of a secured application peer is executed because this happens only one time in the entire runtime peer. When this method is run, also the sub-peers are annexed. In this context the reference of the *AbacAccessManager* is handed over to them, so they can address it with their own *AccessManager* field.

The references of all sub-peers' and the runtime peer's containers are passed to the access manager with the aid of its *RegisterContainers* method. This is necessary because the access manager must be able to query all containers in the runtime peer in order to evaluate the *condition* as discussed in section 6.3.1. Due to the access manager's direct access to all containers, it can simply obtain the rules from the policy peer's PIC. The access manager can also remove the rules from the policy peer‘s PIC when remove rules are received.

The access manager resides in the runtime peer, rather than in the policy peer, in order to decouple the enforcement and execution from the storage of rules. The access control decision is evaluated by the *GetPermission* method.

The *GetPermission* method is called by the already mentioned *EmitRespectingTimeProperties* method, which writes entries to the desired container and tries to run the appropriate wirings. When *GetPermission* returns *true*, the body of *EmitRespectingTimeProperties* is run, otherwise not. The interception of the access manager is placed here, since *EmitRespectingTimeProperties* can be called from several methods. Setting the point of interception one step later, namely shortly before the entries are written to the container, renders the security model vulnerable to DoS attacks, since it would be verified whether the wiring could be run, even if the particular operation were denied.

The *GetPermission* method has the entries, the peer and the wiring name as parameters. All parameters are forwarded parameters from the *EmitRespectingTimeProperties* method. The entries and the peer can be directly used in the *GetPermission* method and the wiring name is used to obtain the container involved in the operation.

The access control decision is whether the stated entries are granted to be written to the container of the stated peer. First, all rules are obtained from the policy peer‘s PIC with the aid of a query against this container. Then the rules are selected by the criteria that are independent of the entries. These criteria are whether the operation's targeted peer and container match the peer and container stated in the rule. The remaining criteria are the subject property template, the scope field and the condition. The entries from a single write operation can possess different subject property chains as the sending peer may forward entries it has received from different peers.

A rule's applicability concerning the subject property template is verified by checking whether the security attribute values of each security attribute of each element from the subject property template is a subset of the corresponding security attribute values of the corresponding element of the entry's subject property chain.

A rule's applicability concerning the scope field is evaluated by checking whether the entry's type is stated in the scope field and whether the additional predicate evaluates to true. The

additional predicate facilitates to use nearly arbitrary data of the entry including coordination data which leads to express dynamic scope fields as described in section 6.3.1. Indeed the flexibility of the additional predicate has the disadvantage that the scope field must be evaluated for every single entry. If the scope field specified only types, the entries could be grouped and the scope field could be evaluated for groups of entries instead of single entries.

The condition evaluates to true or false depending on defined logical conjunctions/disjunctions of condition predicates which are specified queries against defined containers as described in Section 6.3.1. A condition predicate evaluates to either true or false depending on whether the query can be satisfied or not. This depends on the query of the condition predicate and the entries residing in the queried container. Condition predicates are sequentially evaluated and the intermediate result of the logical conjunctions/disjunctions of condition predicates is sequentially evaluated form left to right. For example the condition *[cp1 AND cp2 AND cp3 OR cp4]* with the condition predicates cp1 - cp4 is evaluated by first evaluating the condition predicates whose results are the booleans b1 - b4 and then bring them together by means of the logical conjunctions/disjunctions. The condition's result in this example is the result of (((b1 AND b2) AND b3) OR b4).

The selected rules that are applicable due to the criteria except for the subject property template, the scope field and the condition, are ordered by their application probability. This is achieved by splitting the rules into lists by means of the number of wildcards for the remaining criteria a rule possesses. A rule which possesses more wildcards is more probable to be applicable than a rule with few or no wildcards. If there is a rule in the list with a wildcard for the (whole) subject property template, the scope field and the condition, the operation is granted, because this rule is applicable to all entries. Wildcards for elements of the subject property template are not respected.

The order of inspecting the rules for evaluating an access control decision for each entry is: First the rules with two wildcards are inspected, then the rules with one wildcard are inspected and finally the rules without a wildcard are inspected. Therefore the rules are organized in different lists.

A copy of the entries is stored in a temporary list and when a rule grants access to an entry, the respective entry is deleted from the temporary list and the access control evaluation for this particular entry is skipped. When the temporary list is empty the whole operation is granted, i.e. the *GetPermission* method returns true. When all rules are inspected for an entry and none of them can grants access to the entry, the whole operation is denied, i.e. the *GetPermission* method returns false. The procedure of the *GetPermission* method is depicted as flowchart in Figure 6.9.

As locally written entries have full access within the own runtime peer, the access manager grants entries with the security attribute *LocalAdmin*. This is hard-coded in the *GetPermission* method.

Rule entries must be written by the local user in the first step. When these rules grant rule entries from certain remote users to the policy peer, they can add further rules. The same applies to remove rules. As the access manager has direct access to all containers within the runtime peer and thus to the policy peer's PIC, the access manager executes the deletion of rules when remove rules are received.

6.3.3 Policy Peer

The policy peer holds the rules that are relevant for the runtime peer and its sub-peers. It is a sub-peer itself and exists once per runtime peer. Therefore its creation is triggered by the run method, which is only executed once within the entire runtime peer. In contrast, a peer's startup action is executed for the runtime peer and its sub-peers and is thereby no appropriate trigger for the creation.

An application peer with the name 'POLICY' is produced and inserted into the runtime peer's list of sub-peers. Then two wirings are created whose task is to transport rules and remove rules, respectively, from the runtime peer's PIC to the policy peer. The guard query for the wiring that transfers the rules is constituted as follows:

Operation = QueryOperation.Take

Type = typeof(Rule)

Amount = 0

Relation = QueryRelation.MoreThan

The guard query from the remove rule wiring is similar with the difference that *Type = typeof (RemoveRule)*. These two wirings are inserted in the runtime peer's wirings at first position in order to give the transportation of rules and remove rules priority. With the aid of this architecture rules can be added and removed from the policy peer by writing or sending them to the runtime peer's PIC.

6.3.4 Conclusion of the Implementation

The only requirement that could not be satisfied was the TLS connection between runtime peers. All other requirements are satisfied and some are even exceeded. Due to PeerSpace.NET's support for additional predicates the queries of this security model are more expressive than the queries of the Secure Peer Model. This leads to very expressive scope fields and conditions.

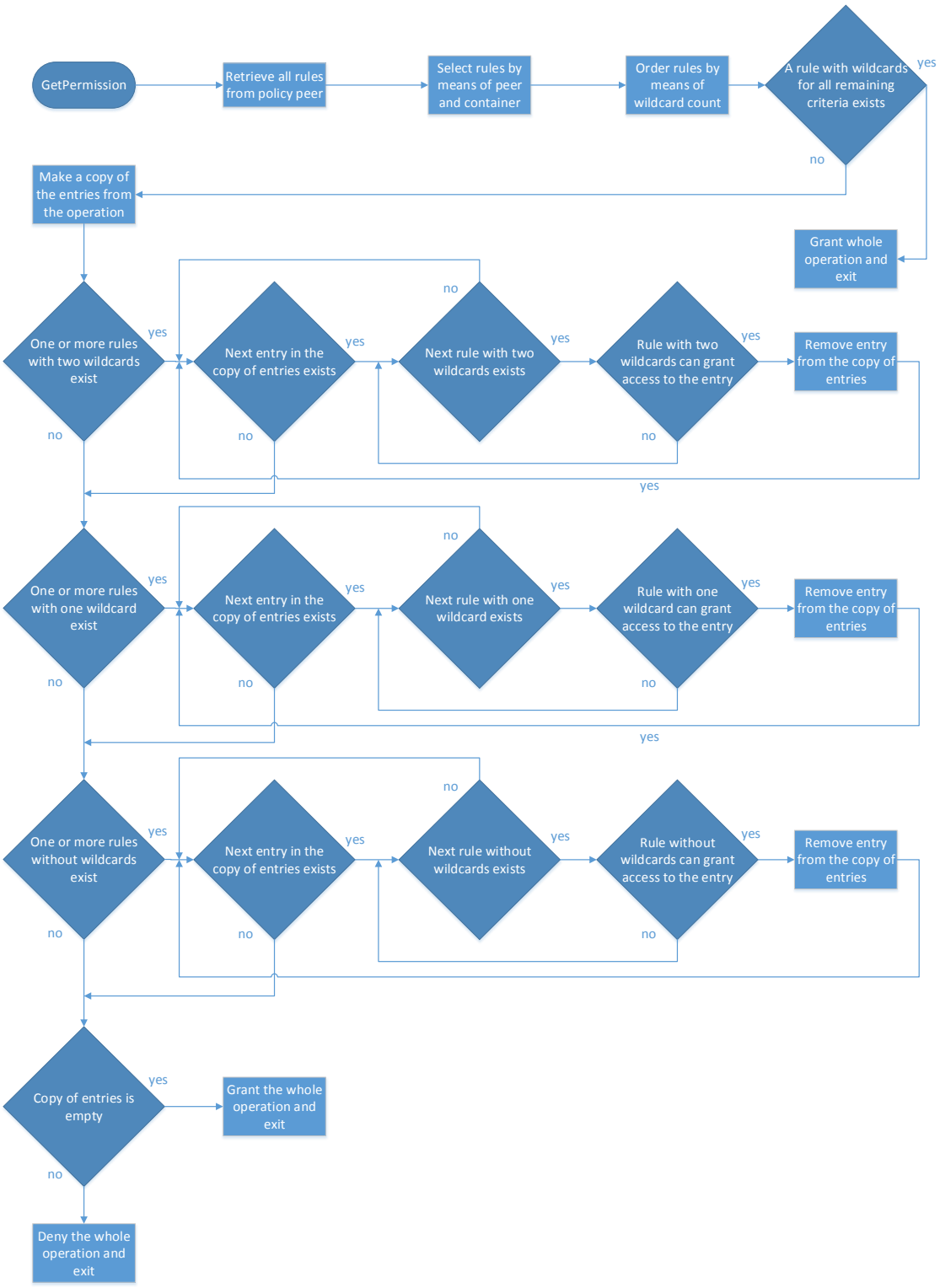

Figure 6.9: Procedure of the *GetPermission* method

CHAPTER 7

Evaluation

The evaluation is an essential part in the methodical approach of creating software, as the functionality of the result has to be proven.

In addition important insights may be gained through the evaluation, and the comparison to other systems can be shown.

This chapter depicts a use case in order to prove the result and a benchmark test to gain insights about the performance impact of the security model. For the comparison to other systems, the features of the created security model are integrated into table 2.1 from the related work chapter.

7.1 Academic Exercise - a Use Case

It has been already shown that the implementation theoretically satisfies the requirements. However its functionality has not been proven yet, which is conducted by means of this use case. It cannot cover all theoretically satisfied requirements, but it demonstrates the fulfillment of several of them.

7.1.1 Basic Setup of the Use Case

The following example depicts the administrative procedure of an academic exercise. In the first step, the explanation is independent from the Peer Model and does not involve any security.

Students, tutors and a supervisor register themselves for the exercise at the lecture server. Then tutors upload exercises to the lecture server which are distributed to all registered students.

After the exercises are done by the students, they send their solutions back to the lecture server, which distributes them to the tutors. They check the solutions and send grading proposals to the lecture server which forwards them to the supervisor of the exercise.

Based on the grading proposals, the supervisor marks the exercises and sends the gradings to the lecture server, which forwards them to the particular students. In this example there will be ten students, two tutors and one supervisor.

The flowchart in Figure 7.1 illustrates the procedure for this use case.

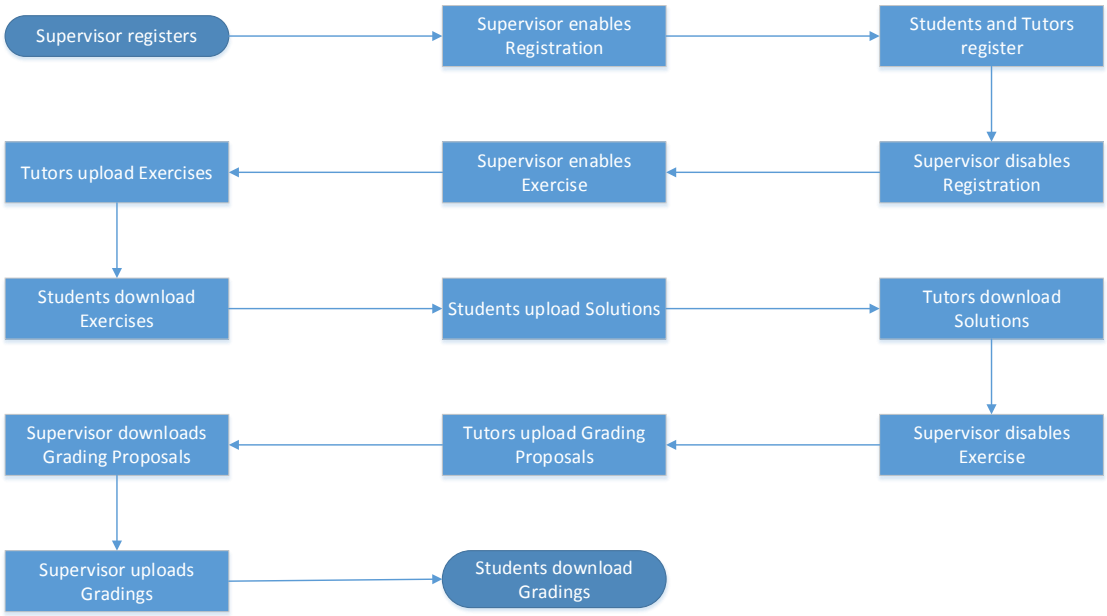

Figure 7.1: Basic procedure of the academic exercise

7.1.2 Academic Exercise Modeled with the PeerSpace.NET without Security

The already examined use case is realized with the PeerSpace.NET as follows. Students, tutors, the supervisor and the lecture server are each implemented as discrete runtime peers. Their interaction and the order of the transmitted entries is depicted in Figure 7.2.

In the following, the entry types and their fields are listed.

- **SuR - Supervisor Registration Entry**
  - Name
  - ID
- **EnR - Enable Registration**
- **StR - Student Registration Entry**
  - Name
  - Matriculation Number
  - Exercise delivered = false (initial state)
- **TuR - Tutor Registration Entry**
  - Name

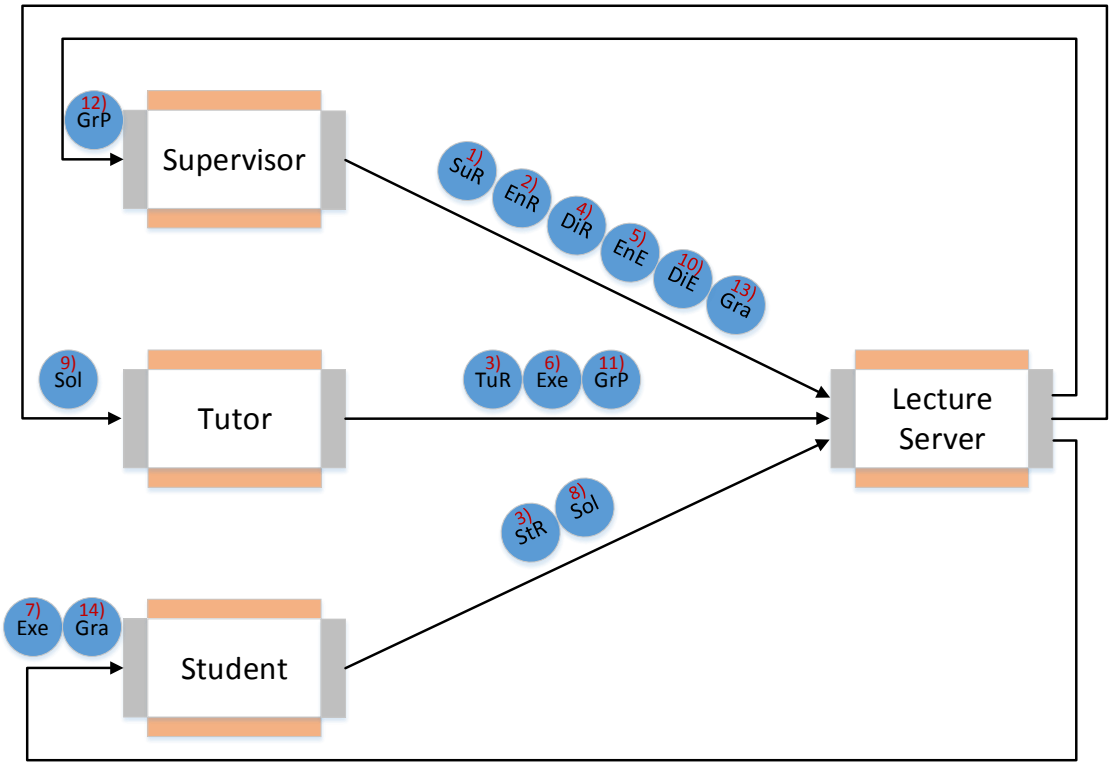

Figure 7.2: Academic exercise modeled with PeerSpace.NET

  - Matriculation Number
  - Assigned Exercises = 0 (initial state)

- **DiR - Disable Registration Entry**
- **EnE - Enable Exercise Entry**
- **Exe - Exercise Entry**
  - Exercise Task
- **Sol - Solution Entry**
  - Matriculation Number
  - Exercise Solution
- **DiE - Disable Exercise Entry**
- **GrP - Grading Proposal Entry**
  - Matriculation Number
  - Grading Proposal

  - Exercise Solution

- **Gra - Grading Entry**
  - Matriculation Number
  - Grading

In this architecture all users communicate only with the lecture server, so they only need to know its address.

The wirings which are directed to the lecture server are pretty straight forward to implement. Thus they are not considered closer. In contrast, the wirings from the lecture server which distribute certain entries by means of the registered users are a little more complex. Thereby the target (address) must be obtained from the coordination data of the particular registration entry and dynamically set in the respective entry via the DEST property. Subsequently it is sent to the set destination.

Such a wiring in the lecture server is the one which distributes the exercises to the students. Therefore its guard reads the exercise entry and takes a student registration where the *Exercise delivered* field states false. Then the service obtains the address from the coordination data of the registration entry and sets therewith the DEST property for the exercise, which is subsequently sent to the desired destination, i.e. the registered student. Further the wiring's service sets the *Exercise delivered* field to true and the *action* writes it back to the PIC. The wiring fires as long as there are student registration entries where the *Exercise delivered* field states false. Figure 7.3 illustrates this wiring that distributes the exercises to students according to the student registrations.

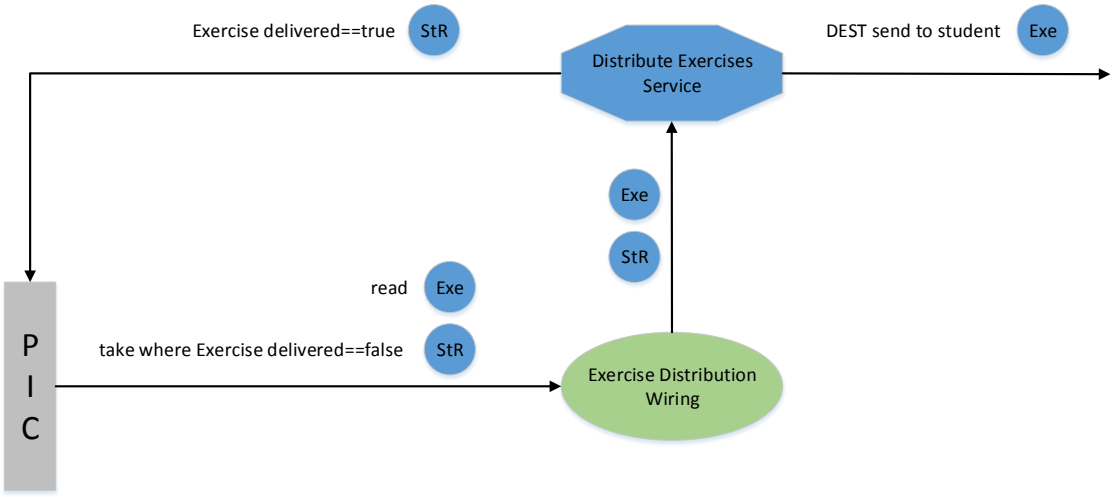

Figure 7.3: Wiring that distributes the exercises to the students

The solutions are forwarded from the lecture server to the a tutor in a similar way. A wiring takes a solution and a tutor registration entry, where the count of the *Assigned exercises* field is less than five. Next the according service reads the address of the tutor from the coordination

data of his/her registration entry and sends the solution with the aid of the DEST property to the tutor. Then the service increases the tutor registration entry's count in the *Assigned exercises* field and writes it back to the PIC. In this way five solutions are assigned to each tutor.

The forwarding of the grading proposal to the supervisor occurs alike the distributions of solutions, with the difference that the task does not have to be divided.

Finally, the wiring which distributes the grading to the respective student is a little more complex to realize. The guard query takes a grading and reads all student registrations. The service makes a query against the entry collection in order to obtain the address of the student's registration which matches the grading by the matriculation number. This is actually not the service's task, but the guard did not know the student's matriculation number before the wiring has fired. Next the service sets the DEST property in the grading entry which is subsequently sent to the respective student peer.

As the use case without security has been successfully realized with the PeerSpace.NET, we can now focus on the access control. In the first step the explanation will be general, i.e. independent of the PeerSpace.NET.

7.1.3 General Access Control for the Academic Exercise

The procedure starts when a supervisor registers at the lecture server which grants him/her the required access on the lecture server to control the exercise. Next the supervisor enables the registration for this exercise for students and for tutors.

Students register themselves at the lecture server. This shall only be possible when the maximum registration count of students has not been reached and the registration has started and not ended yet. Registered students accept exercises and gradings from the lecture server.

The registration for tutors is likewise the one of the students. Tutors accept solutions from the lecture server.

When the supervisor ends the registration and starts the exercise, nobody can register any more and the registered tutors and students are granted to upload exercises and solutions, respectively. This is possible as long as the supervisor has not ended the exercise.

When the exercise ends, the tutors are permitted to upload grading proposals to the lecture server, which are forwarded to the supervisor and build the basis for the grading. The supervisor uploads the gradings to the lecture server from where it is sent to the respective students.

7.1.4 Access Control for the Academic Exercise with the PeerSpace.NET

Let us now focus on the implementation with the secured PeerSpace.NET. The particular rules are examined in the context of this use case.

First a supervisor registers itself at the lecture server by sending a *SupervisorRegistration* entry to it. This is only granted when a supervisor sends its own registration (scope field) and no other supervisor has been registered yet (condition). Note that $ID refers to the sender's id obtained from the entry's subject property chain.

- **Lecture Server Rule 1**
  - Subject Property Template [0]: Role = Supervisor

  - Resources: Runtime Peer PIC
  - Operation: Write
  - Condition: [¬, Amount = 1, Type = SupervisorRegistration]
  - Scope field: Type = SupervisorRegistration, Additional Predicate = (ID = $ID)
  - Effect: Permit

The next rule grants the registered supervisor (condition) write access to the lecture server's PIC for the entries of the types stated in the scope field.

- **Lecture Server Rule 2**
  - Subject Property Template [0]: Role = Supervisor
  - Resources: Runtime Peer PIC
  - Operation: Write
  - Condition: [Amount = 1, Type = SupervisorRegistration, Additional Predicate = (ID = $ID)]
  - Scope field: Type = EnableRegistration, DisableRegistration, EnableExercise, DisableExercise, Grading
  - Effect: Permit

The lecture server accepts student registration entries when the registration has been started and not ended yet and less than ten registration have been done. This is stated in the rule's condition. The dynamic scope field determines that students can only sent registrations for themselves which is validated by the matriculation number.

- **Lecture Server Rule 3**
  - Subject Property Template [0]: Role = Student
  - Resources: Runtime Peer PIC
  - Operation: Write
  - Condition: [Amount = 1, Type = EnableRegistration] AND [¬, Amount = 1, Type = DisableRegistration] AND [¬, Amount = 10, Type = StudentRegistration]
  - Scope field: Type = StudentRegistration, Additional Predicate = (MNR = $MNr)
  - Effect: Permit

The following rule grants registrations for tutors.

- **Lecture Server Rule 4**
  - Subject Property Template [0]: Role = Tutors
  - Resources: Runtime Peer PIC

  - Operation: Write
  - Condition: [Amount = 1, Type = EnableRegistration] AND [¬, Amount = 1, Type = DisableRegistration] AND [¬, Amount = 2, Type = TutorRegistration]
  - Scope field: Type = TutorRegistration, Additional Predicate = (MNR = $MNr)
  - Effect: Permit

The next rule grants the registered tutors (condition) the upload of exercises (scope field) to the lecture server, under the premise that the exercise has started and not ended (condition).

- **Lecture Server Rule 5**
  - Subject Property Template [0]: Role = Tutors
  - Resources: Runtime Peer PIC
  - Operation: Write
  - Condition: [Amount = 1, Type = TutorRegistration, Additional Predicate = (MNR = $MNr)] AND [Amount = 1, Type = EnableExercise] AND [¬, Amount = 1, Type = DisableExercise]
  - Scope field: Type = Exercise
  - Effect: Permit

Further, registered tutors are allowed to upload grading proposals when the exercise has ended. This rule acts similar to the former one, but possesses a different scope field and a slightly different condition.

- **Lecture Server Rule 6**
  - Subject Property Template [0]: Role = Tutors
  - Resources: Runtime Peer PIC
  - Operation: Write
  - Condition: [Amount = 1, Type = TutorRegistration, Additional Predicate = (MNR = $MNr)] AND [Amount = 1, Type = DisableExercise]
  - Scope field: Type = Grading Proposal
  - Effect: Permit

Following rule grants registered students (condition) the upload of solutions (scope field) to the lecture server, under the premise that the exercise has started and not ended (condition).

- **Lecture Server Rule 7**
  - Subject Property Template [0]: Role = Student
  - Resources: Runtime Peer PIC

  - Operation: Write
  - Condition: [Amount = 1, Type = StudentRegistration, Additional Predicate = (MNR = $MNr)] AND [Amount = 1, Type = EnableExercise] AND [¬, Amount = 1, Type = DisableExercise]
  - Scope field: Type = Solution, Additional Predicate = (MNR = $MNr)
  - Effect: Permit

The following rule is located in the student peers. It grants the lecture server write access to a student peer for entries of the types *Exercise* and *Grading*.

- **Student Rule 1**
  - Subject Property Template [0]: Role = Lecture Server
  - Resources: Runtime Peer PIC
  - Operation: Write
  - Condition: None
  - Scope field: Type = Exercise, Grading
  - Effect: Permit

The next rule is located in the tutor peers and is similar to the previous rule but possesses a different scope field.

- **Tutor Rule 1**
  - Subject Property Template [0]: Role = Lecture Server
  - Resources: Runtime Peer PIC
  - Operation: Write
  - Condition: None
  - Scope field: Type = Solution
  - Effect: Permit

The supervisor accepts only grading proposals from the lecture server which have been uploaded by a tutor to the lecture server. Thus, the lecture server is stated as direct sender and the tutor is stated as the first indirect sender in the subject property template of the following rule. This ensures that the grading proposal was issued by a tutor instead of, e.g., a malicious student.

- **Supervisor 1**
  - Subject Property Template [0]: Role = Lecture Server
  - Subject Property Template [1]: Role = Tutor
  - Resources: Runtime Peer PIC

- Operation: Write
- Condition: None
- Scope field: Type = GradingProposal
- Effect: Permit

Figure 7.4 illustrates the procedure of the secured academic exercise and states the correlation between the granting rules and the respective process steps.

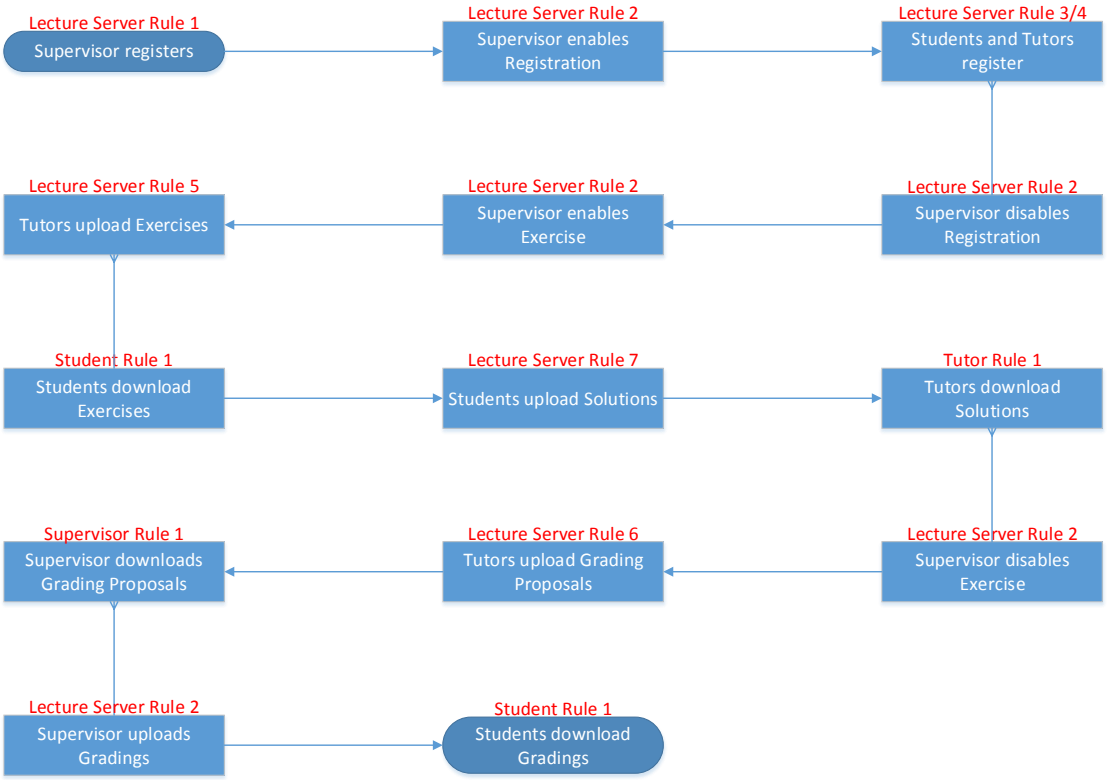

Figure 7.4: Process steps with the corresponding rules

7.1.5 Satisfied Requirements Demonstrated with the Use Case

Every peer has its own policy, thus enables that, e.g., the lecture server peer has a complete different policy than a student peer. The use case demonstrates the expressiveness of the implemented security model by employing rules where complex scope fields and/or conditions are specified. For example, “Lecture Server Rule 3” allows students to send their own registrations (dynamic scope field) under the condition that the registration is enabled and not disabled and that less than ten students are already registered. “Lecture Server Rule 7” grants registered students (dynamic condition) to upload their solutions (dynamic scope field) under the condition that the exercise is enabled and not disabled. The supervisor peer accepts grading proposals

from the lecture server that are sent on behalf of a tutor. This is defined in “Supervisor Rule 1” which demonstrates the authorization for delegated entries.

Some features that have not been demonstrated are in the following discussed in the context of the use case. When, e.g., the number of allowed students is increased to fifteen and some students are already registered, it is useful to update the policy during run time, which is feasible with the implemented security model. A further feature of the security model is the delegation for the possibly remote administration of a peer's policy to certain users. This feature could be used to enable the supervisor to change the policy, e.g., when the number of allowed students is increased. The supervisor could achieve this task from home. The security model supports wildcards for, among others, an arbitrary number of subject property chain elements. Thus, e.g., a rule can be defined which grants a solution entry, that is sent from the lecture server peer and has its origin at a student peer, access to the supervisor peer, regardless of the intermediate peers (e.g. tutor peers).

7.1.6 Gained Findings from the Use Case

The rules from the use case are pretty straight forward to define and manage. The implemented access control facilitate a fine-grained policy which would not be feasible with simple access control lists (ACLs). For example, the rule *Lecture Server Rule 5* that grants only registered tutors under the condition that the exercise has started and not ended exceeds the expressiveness of ACLs. Further, the rule *Supervisor 1* demonstrates the expressiveness of the implemented access control by also including the identity of the indirect sender to state whether the rule is applicable. It was feasible to successively implement the use case as no security was enabled and only the coordination of the use case was modeled in the first step. When security was enabled (during start-up) the policies could be dynamically managed with ease.

7.2 Benchmark Test

The benchmarks were executed on a standard laptop (CPU Intel 3610QM, 2.30 GHz, Windows 8.1) with the *Stopwatch* shipped with .NET. Each benchmark was executed five times and the average times are depicted in Table 7.1. The test scenario is as follows: Runtime peer A sends 100, 1000 and 10000 entries of the type string to runtime peer B. Both runtime peers are hosted on the same computer, thus there is no network latency in this benchmark test. The authentication of entries is conducted for every access control. The entries are generated in advance and written to runtime peer A which sends all entries at once to runtime peer B with the aid of a wiring, whose service starts the *Stopwatch*. Runtime peer B possesses a wiring whose service stops the *Stopwatch*.

The benchmarks were executed without access control (No AC), with one simple rule (1 SR), ten simple rules (10 SR), one complex rule (1 CR) and ten complex rules (10 CR). The simple rules define only the subject property template and the complex rules define the subject property template, a scope field with additional predicate and a condition with two condition predicates. Note that all unspecified fields of a rule are defined as wildcards. When the benchmark test run with one rule, this rule granted access to the sent entries. When the benchmark test run with

ten rules, only one of them granted access to the sent entries. To obtain results that correspond neither to best nor to worst case scenarios, the granting rule was positioned in the middle of the ten rules.

The first column in Table 7.1 shows the count of the sent entries. The other columns state the elapsed time in milliseconds.

Entries	No AC	1 SR	10 SR	1 CR	10 CR
100	78	567	580	617	701
1000	168	737	743	951	1734
10000	1569	3148	3252	5165	13033

Table 7.1: Benchmark tests with different counts and complexity of rules

7.2.1 Interpretation of the Benchmark Tests

When 100 entries are sent to runtime peer B the time elapsed until they are authorized does not differ much between the different employed rules. Thus the authentication process where the receiving runtime peer queries the sender's security attributes from the identity provider forms the most overhead in this case.

There is nearly no time difference when one or ten simple rules are employed. That is because these rules possess all the information needed for an access control decision and when the rules are in the cache of the access manager once, there is not much difference between evaluating one or five rules.

In contrast, the evaluation of complex rules where a scope field and a condition are specified takes much more time. That is because information must be gathered in order to make an access control decision. For the evaluation of the scope field only information from the entry whose access control decision is made must be gathered, but for the evaluation of the condition, query operations in the Peer Space are needed. This causes much overhead which reflects the field in Table 7.1 where 10000 entries are authorized with one of ten complex rules.

Note that in real world scenarios there is a network latency and some computations with the entries are executed which takes time. This puts the results of the benchmark tests into perspective.

The findings of the benchmark tests are that complex rules lower the performance. These kind of rules should only be employed when necessary. Further, when entries that are granted by a complex rule can also be granted by a (partly redundant) simple rule, this simple rule should also be employed as it grants access with less overhead than the complex rule.

7.3 Comparison to the Analyzed Systems from the Related Work

Several middlewares were compared concerning their security features in the related work chapter. Table 7.2 depicts the comparison table which also includes the implemented security model for the PeerSpace.NET (SPSN) to illustrate its positioning.

	XVSM	Hermes	SMEPP	TuCSon	WCF	SPSN
RBAC	+	+	−	+	+	+
ABAC	+	−	−	−	+	+
Content-aware rules	+	+	−	+	∼	+
Context-aware rules	+	+	−	+	∼	+
Authorization for indirect sender	∼	−	−	≃	≃	+
Wildcard support for indirect sender	−	−	−	−	−	+
Dynamic policies	+	+	−	+	+	+
Remote policy changes	+	+	−	+	+	+
Administration delegation	+	+	−	+	+	+
Bootstrapped architecture	+	+	∼	+	−	+
Transparency	+	+	+	+	−	+
Scalability	∼	+	∼	∼	+	∼

Table 7.2: Positioning of the implemented security model SPSN

+: supported
∼: supported with limitations
≃: supported with major limitations
−: not supported

The implemented security model fulfills all requirements and satisfies all points stated in Table 7.2. Content-aware rules are feasible due to a rule's scope field and context-aware rules are feasible due to a rule's condition. The (dynamic) scope field may involve data from the user and the (dynamic) condition may depend on the entry the rule is evaluated for and on data of the user who sent the entry. Rules can be added and removed possibly from remote during run time as rules and remove rules are realized as entries and transmitted with the mechanisms of the PeerSpace.NET, i.e. the security model's architecture is also bootstrapped. The administration of the policy can be delegated to a certain user with the aid of rules defining this user access to the own policy peer.

The implemented security model is the most expressive model in Table 7.2 due to *Authorization for indirect sender* and *Wildcard support for indirect sender* because it facilitates to involve the indirect senders of an arbitrarily long delegation chain and supports wildcards for an arbitrary and possibly unknown count of elements of the subject property chain.

CHAPTER 8

Future Work

During the design and implementation of the security model we found some future work which is stated in this chapter.

Unauthorized operations are denied and logged, which requires computing power. This renders the system vulnerable for DoS attacks. To reduce this vulnerability a mechanism which cleverly prevents the logging of every denied operation should be implemented.

The current implementation of the security model uses an identity provider from a single organization. The support of identity providers from diverse organizations and the integration of an existing identity provider instead of the file-based solution would be beneficial for the practical employment of the system.

Entries are authenticated by attaching the sender's security attributes. Assume entries reside in the PeerSpace.NET and are authenticated with the security attributes from a user that is meanwhile kicked from the system. These entries possess obsolete security attributes and consequently the access control applied to them may be incorrect. Therefore a better approach would be to associate the entries with the id of the sender and request the up-to-date security attributes for every access control decision.

A pool of prevalent security patterns whereof the appropriate rules are derived and automatically created would be a handy feature which facilitates the administration of the access control.

Further future work requires an enhancement of the PeerSpace.NET, namely the possibility to add sub-peers and wirings to a peer from remote and by different users. Thus a multi-user concept would be realized where different parts of peers may belong to different users. Future work would be to adapt the subject property template for this multi-user concept, whereby the users may be registered at different organizations and possibly indirectly authenticated. Access control for read and take operations would also be necessary to secure the enhanced PeerSpace.NET.

CHAPTER 9

Conclusion

The goal of this thesis was the design and implementation of a security model for the PeerSpace.NET coordination middleware. Authentication and authorization methods were studied, as well as middlewares, the Peer Model and the PeerSpace.NET. With the aid of the gained knowledge the requirements for the security model were gathered. As the PeerSpace.NET is a P2P middleware, where no centralized server exists which could conduct the authentication, an appropriate mechanism therefore had to be found. The solution was a stand-alone identity provider with the capability to administrate user registrations.

The main requirements concerning the authorization were the possibility to express fine-grained rules which may also involve the identity of indirect senders. Further, peer owners should be able to define and enforce their own policy or possibly delegate this task to certain users.

On the basis of the requirements and the knowledge about the PeerSpace.NET the design was created which depicts the architecture of the security model and describes the necessary procedures. On its basis the implementation of the security model for the PeerSpace.NET was carried out.

The evaluation of the implementation comprises a theoretical validation of the requirements' satisfaction, a use case, a benchmark test and a comparison to other middlewares. The use case demonstrates the functionality of the security model and shows the application of fine-grained policies which use content- and context information for the evaluation of access control. The benchmark tests indicate that the security model scales well for simple rules. The comparison depicts the security features in contrast to other middlewares.

Thus the evaluation indicates the successful implementation of the security model.

Bibliography

[1] David Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, 1985.

[2] eva Kühn, Johannes Riemer, and Gerson Joskowicz. Xvsm (extensible virtual shared memory) architecture and application. Technical report, TU-Vienna E185/1, 2005.

[3] eva Kühn, Stefan Craß, Gerson Joskowicz, Alexander Marek, and Thomas Scheller. Peer-Based Programming Model for Coordination Patterns. In *Coordination Models and Languages*, pages 121–135. Springer, 2013.

[4] Dominik Rauch. Peerspace.net: Implementing and evaluating the peer model with focus on api usability. Master's thesis, 2014.

[5] Joan Arnedo-Moreno and Jordí Herrera-Joancomartí. A survey on security in jxta applications. *Journal of Systems and Software*, 82(9):1513–1525, 2009.

[6] Chander Diwakar. Security threats in peer to peer networks. *Journal of Global Research in Computer Science*, 2(4), 2011.

[7] Fred Schneider. Least privilege and more. In *Computer Systems*, pages 253–258. Springer, 2004.

[8] Xuhui Ao and Naftaly Minsky. Flexible regulation of distributed coalitions. In *Computer Security–ESORICS 2003*, pages 39–60. Springer, 2003.

[9] Stefan Craß, eva Kühn, and Gerson Joskowicz. A decentralized access control model for dynamic collaboration of autonomous peers. *11th EAI International Conference on Security and Privacy in Communication Networks (SecureComm)*, 2015. Accepted for publication.

[10] Roger Needham and Michael Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, December 1978.

[11] Dick Hardt. The oauth 2.0 authorization framework. 2012.

[12] David Recordon and Brad Fitzpatrick. Openid authentication 1.1. *Finalized OpenID Specification, May*, 2006.

[13] Dick Hardt, Johnny Bufu, and Josh Hoyt. Openid attribute exchange 1.0-final. *at, Dec*, 5:11, 2007.

[14] HwanJin Lee, InKyung Jeun, Kilsoo Chun, and Junghwan Song. A new anti-phishing method in openid. In *Emerging Security Information, Systems and Technologies, 2008. SECURWARE'08. Second International Conference on*, pages 243–247. IEEE, 2008.

[15] Jothy Rosenberg and David Remy. *Securing Web Services with WS-Security: Demystifying WS-Security, WS-Policy, SAML, XML Signature, and XML Encryption*. Pearson Higher Education, 2004.

[16] Clifford Neuman and Theodore Ts' O. Kerberos: An authentication service for computer networks. *Communications Magazine, IEEE*, 32(9):33–38, 1994.

[17] Rolf Oppliger, Ralf Hauser, and David Basin. Ssl/tls session-aware user authentication. *Computer*, (3):59–65, 2008.

[18] Simon Godik, Anne Anderson, Bill Parducci, Polar Humenn, and Sekhar Vajjhala. Oasis extensible access control 2 markup language (xacml) 3. Technical report, Tech. rep., OASIS, 2002.

[19] Ravi Sandhu and Pierangela Samarati. Access control: principle and practice. *Communications Magazine, IEEE*, 32(9):40–48, 1994.

[20] Ravi Sandhu. Separation of duties in computerized information systems. In *DBSec*, pages 179–190. Citeseer, 1990.

[21] David Ferrialo, Janet Cugini, and Richard Kuhn. Role based access control: Features and motivation. In *Computer Security Application Conference*, 1995.

[22] Eric Yuan and Jin Tong. Attributed based access control (abac) for web services. In *Web Services, 2005. ICWS 2005. Proceedings. 2005 IEEE International Conference on*. IEEE, 2005.

[23] eva Kühn, Richard Mordinyi, Hannu-Daniel Goiss, Thomas Moser, Sandford Bessler, and Slobodanka Tomic. Integration of Shareable Containers with Distributed Hash Tables for Storage of Structured and Dynamic Data. In *Complex, Intelligent and Software Intensive Systems, 2009. CISIS '09. International Conference on*, pages 866–871, 2009.

[24] eva Kühn and Vesna Sesum-Cavic. A space-based generic pattern for self-initiative load balancing agents. In *Engineering Societies in the Agents World X*, pages 17–32. Springer, 2009.

[25] eva Kühn, Alexander Marek, Thomas Scheller, Vesna Sesum-Cavic, Michael Vögler, and Stefan Craß. A space-based generic pattern for self-initiative load clustering agents. In *Coordination Models and Languages*, pages 230–244. Springer, 2012.

[26] Stefan Craß, Jürgen Hirsch, eva Kühn, and Vesna Šešum-Čavić. An Adaptive and Flexible Replication Mechanism for Space Based Computing. In *8th International Conference of Software and Data Technology (ICSOFT-EA)*, Reykjavik, Iceland, July 29-31 2013. INSTICC Press.

[27] Stephen Bourne. *The UNIX system*, volume 247. Addison-Wesley Reading, Massachusetts, 1983.

[28] 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, New York, NY, USA, 2012. ACM.

[29] Daniel Wutke, Daniel Martin, and Frank Leymann. Facilitating complex web service interactions through a tuplespace binding. In *Distributed Applications and Interoperable Systems*, pages 275–280. Springer, 2008.

[30] 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.

[31] 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.

[32] Stefan Craß, Tobias Dönz, Gerson Joskowicz, and eva Kühn. A coordination-driven authorization framework for space containers. *Seventh International Conference on Availability, Reliability and Security (ARES)*, 4(1):76–97, 2012.

[33] Peter Robert Pietzuch. *Hermes: A scalable event-based middleware*. PhD thesis, University of Cambridge Cambridge, UK, 2004.

[34] 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*, pages 1–8. ACM, 2003.

[35] 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 (TISSEC)*, 5(4):492–540, 2002.

[36] Fabrizio Benigni, Antonio Brogi, Jean-Louis Buchholz, Jean-Marie Jacquet, Julien Lange, and Razvan Popescu. Secure p2p programming on top of tuple spaces. In *Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 54–59. IEEE, 2008.

[37] 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.

[38] Rafael Caro Benito, Daniel Garrido Márquez, Pierre Plaza Tron, Rodrigo Raomán Castro, Nuria Sanz Martín, and José Luis Serrano Martín. Smepp: A secure middleware for embedded p2p. *Proceedings of ICT-MobileSummit*, 9, 2009.

[39] 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.

[40] Amy Murphy, Gian Pietro Picco, and Gruia-Catalin Roman. Lime: A coordination model and middleware supporting mobility of hosts and agents. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(3):279–328, 2006.

[41] Andrea Omicini and Franco Zambonelli. Coordination for internet application development. *Autonomous Agents and Multi-agent systems*, 2(3):251–269, 1999.

[42] Satoshi Matsuoka and Satoru Kawai. Using tuple space communication in distributed object-oriented languages. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '88, pages 276–284, New York, NY, USA, 1988. ACM.

[43] Marco Cremonini, Andrea Omicini, and Franco Zambonelli. Coordination and access control in open distributed agent systems: The tucson approach. In *Coordination Languages and Models*, pages 99–114. Springer, 2000.

[44] 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.

[45] Microsoft. What is windows communication foundation. `https://msdn.microsoft.com/en-us/library/ms731082(v=vs.110).aspx`, Aug 2015. Accessed: 2015-08-30.

[46] Microsoft. Introduction to iis. `https://www.iis.net/learn/get-started/introduction-to-iis`, Aug 2015. Accessed: 2015-08-30.

[47] Juval Lowy. *Programming WCF Services: Mastering WCF and the Azure AppFabric Service Bus*. O'Reilly Media, Inc., 2010.

[48] Microsoft. [ms-nlmp]: Nt lan manager (ntlm) authentication protocol. `https://msdn.microsoft.com/en-us/library/cc236621.aspx`, Aug 2015. Accessed: 2015-08-30.

[49] David Solo, Russell Housley, and Warwick Ford. Internet x. 509 public key infrastructure certificate and crl profile. 1999.

[50] Wrox Author Team, Brian Loesgen, Andreas Eide, Mike Clark, Chris Miller, Matthew Reynolds, Robert Eisenberg, Bill Sempf, Srinivasa Sivakumar, Mike Batongbacal, et al. *Professional ASP. net web services*. Wrox Press Ltd., 2001.

[51] Microsoft. Windows communication foundation security. `https://msdn.microsoft.com/en-us/library/ms732362(v=vs.110).aspx`, Aug 2015. Accessed: 2015-08-30.

[52] Microsoft. Access control mechanisms. `https://msdn.microsoft.com/en-us/library/ms733106(v=vs.110).aspx`, Aug 2015. Accessed: 2015-08-30.

[53] Microsoft. Managing claims and authorization with the identity model. `https://msdn.microsoft.com/en-us/library/ms729851(v=vs.110).aspx`, Aug 2015. Accessed: 2015-08-30.

[54] Microsoft. Windows authorization manager. `https://msdn.microsoft.com/en-us/library/bb897401.aspx`, Aug 2015. Accessed: 2015-08-30.

[55] Thomas Scheller. Xcoordination application space. `https://xcoappspace.codeplex.com/`, Aug 2015. Accessed: 2015-08-31.

[56] Andrew Troelsen. *Pro C# 5.0 and the .NET 4.5 Framework*. Apress, Berkely, CA, USA, 6th edition, 2012.

[57] Charles Petzold. *Programming Windows with C# (Core Reference)*. Microsoft Press, 2001.

[58] Łukasz Gąsior. *ReSharper Essentials*. Packt Publishing Ltd, 2014.

[59] Apache Software Foundation. Apache log4net. `https://logging.apache.org/log4net/`, Aug 2015. Accessed: 2015-08-25.

[60] Donald Eastlake 3rd and Paul Jones. Us secure hash algorithm 1 (sha1). Technical report, 2001.