Die approbierte Originalversion dieser Dissertation ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).

FAKULTÄT FÜR !INFORMATIK

DISSERTATION

Managing Complex and Dynamic Software Systems with Space-Based Computing

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Doktors der technischen Wissenschaften unter der Leitung von

Ao. Univ.Prof. Dr. eva Kühn
185-1
Institut für Computersprachen,
Programmiersprachen und Übersetzer

und
Ao. Univ.Prof. Dr. Stefan Biffl

eingereicht an der Technischen Universität Wien
Fakultät für Informatik
von

Dipl.-Ing. Mag.rer.soc.oec. Richard Mordinyi
9825381
Am Schloßberg 13
A-2191 Pellendorf

Wien,17.05.2010

Verfasser	Betreuerin	Zweitbetreuer

„Wer das Ziel kennt, kann entscheiden; wer entscheidet, findet Ruhe; wer Ruhe findet, ist sicher; wer sicher ist, kann überlegen; wer überlegt, kann verbessern."

Konfuzius (551-479 v.Chr.), chin. Philosoph

Danksagung

Zu Beginn möchte ich mich bei meiner Betreuerin Ao. Univ. Prof. Dr. eva Kühn für die langjährige, stets positive Zusammenarbeit und die wichtigen Ideen und Perspektiven, die diese Arbeit maßgeblich vorangetrieben haben, bedanken. Außerdem möchte ich meinem Zweitbetreuer Ao. Univ. Prof. Dr. Stefan Biffl vom Institut für Softwaretechnik und interaktive Systeme für die unkomplizierte Kooperation, sowie für die neuen Impulse, die er dieser Arbeit gegeben hat, danken. Überdies möchte ich die vielen gemeinsamen Überlegungen und Arbeitsstunden mit meinem Kollegen Thomas Moser anerkennen.

Darüber hinaus möchte ich mich bei meinen Kolleginnen und Kollegen am Institut für Computersprachen und am Institut für Softwaretechnik und interaktive Systeme für die konstruktive Zusammenarbeit und Unterstützung der vergangenen Jahre bedanken, vor allem bei Institutsvorstand Prof. Dr. Jens Knoop für die großzügige Unterstützung in Studienangelegenheiten; außerdem bei Amin Anjomshoaa, Marcus Mor, Martin Murth, Alexander Schatten, Fabian Schmied, Johannes Riemer, und Prof. Dr. A Min Tjoa. Im Bereich der Zusammenarbeit mit Industrie- und Forschungspartnern möchte ich mich besonders bei Alexander Mikula (Frequentis), Sandford Bessler (ftw), Slobodanka Tomic (ftw), und Thomas Frühbeck (Telekom Austria) für die praktisch relevanten Herausforderungen und Evaluierungen bedanken. Ferner möchte ich den Studierenden am Institut, v. a. Martin Barisits, Stefan Craß, Severin Ecker, Hannu-Daniel Goiss, Rene Hais, László Keszthelyi, Michael Lafite, Lukas Lechner, Mario Lang, Bernhard Löwenstein, Manuel Maly, Michael Pröstler, und Christian Schreiber für ihre wertvollen Beiträge im Rahmen von Praktika oder Diplomarbeiten der vergangenen Jahre danken, die für die Erstellung dieser Arbeit unerlässlich waren.

Abschließend möchte ich auch meine Familie, allen voran meine Eltern, die mir dieses Studium erst ermöglicht haben, dankend erwähnen. Ganz besonderer Dank gebührt meiner Ehefrau, die mich in meinen Vorhaben stets geduldig unterstützt und aus deren Beisammensein ich viel Kraft schöpfe.

Eidesstattliche Erklärung

Dipl.-Ing. Mag.rer.soc.oec. Richard Mordinyi

Am Schloßberg 13/1

2191 Pellendorf

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

Wien, 17. Mai 2010 ____________________________

Abstract

Software systems are usually composed of distributed and heterogeneous application components representing higher-level business goals, and a middleware part abstracting the complexity concerns related to network and distribution.

In the course of developing complex software systems software developers have to deal with interacting application components and changing business requirements. The message-passing paradigm is a common concept allowing application components to interact with each other. But even asynchronous message-oriented middleware technologies are not suitable for complex coordination requirements since the processing and state of coordination have to be handled explicitly by the application component, thus increasing its complexity. Data-driven frameworks, like tuple spaces, support the coordination of application components, but have a limited number of coordination policies. Therefore, with respect to more complex coordination requirements application components still need to implement coordination functionality that is not directly supported by the coordination framework.

Middleware frameworks usually represent a specific architectural style and there is a dependency between application components and architectural style. In case a new business requirement demands the implementation of other architectural styles, the combinations of those styles further introduces additional cognitive complexity. Consequently, instead of a stable set of architectural concepts for effectively managing complexity concerns, the number of concepts a software developer has to work with explicitly increases with the size and degree of evolution of the system.

This thesis proposes the so called Space-Based Computing (SBC) paradigm to support software developers managing complexity issues regarding interaction requirements and agility of software architectures. The SBC paradigm defines an architectural style that flexibly combines and abstracts the properties of several architectural styles and extends them by sophisticated coordination models. In contrast to traditional coordination frameworks the approach supports exchangeable coordination models and regardless of the evolutionary degree or state of the system, it offers software developers a stable and limited number of architectural concepts to work with. The approach is evaluated in several industrial application domains: in air traffic management, production automation, and intelligent transportation regarding feasibility, effort, robustness, performance, scalability, and usability. The evaluation was carried out by means of prototype implementations, studies, benchmarks, and theoretical proofs. The results show a higher coordination efficiency, improved robustness against changing requirements, simplified realization of business requirements, and reduced complexity in applications.

Zusammenfassung

Software-Systeme bestehen meistens aus verteilten und heterogenen Applikationskomponenten, die Geschäftsziele repräsentieren, und aus einer Middlewareschicht, die die Komplexität von Netzwerken und Verteilung abschirmt.

Im Zuge der Entwicklung von komplexen Software-Systemen müssen sich Softwareentwickler interagierenden Applikationskomponenten und sich ändernden Geschäftsanforderungen stellen. Das nachrichtenbasierte Paradigma ist ein bekanntes Konzept, um die Kommunikation zwischen Applikationskomponenten zu ermöglichen. Es sind jedoch nicht einmal asynchrone nachrichtenbasierte Middlewaretechnologien in der Lage, komplexe Koordinationsanforderungen zu erfüllen, da die Applikationskomponente die Abarbeitung und den Zustand der laufenden Koordination verwalten müssen. Dies steigert die Komplexität der Applikation. Datengesteuerte Frameworks, wie der Tuple Space, hingegen unterstützen die Koordination von Applikationskomponenten, weisen aber eine limitierte Anzahl an Koordinationsformen auf, die eine zusätzliche Komplexität einführen, sobald komplexe Koordinationsanforderungen erfüllt werden müssen.

Die vorliegende Dissertation schlägt das sogenannte Space-Based Computing (SBC) Paradigma vor, um Softwareentwickler bei der Handhabung der Komplexität von Interaktions- und Agilitätsanforderungen zu unterstützen. Das SBC Paradigma ist ein Architekturstil, der flexibel die Eigenschaften von unterschiedlichen Architekturstilen kombiniert und abstrahiert, und diese um Koordinationsmodelle anreichert. Im Gegensatz zu traditionellen Koordinationsframeworks ermöglicht SBC die Austauschbarkeit von Koordinationsmodellen. Weiter ermöglicht es Softwareentwicklern unabhängig vom Zustand oder Entwicklungsgrad des Systems, mit einer gleichbleibenden Anzahl an Konzepten zu arbeiten. Der vorgestellte Ansatz wurde hinsichtlich Umsetzbarkeit, Aufwand, Robustheit, Leistung, Skalierbarkeit und Bedienbarkeit in unterschiedlichen Domänen eingesetzt und evaluiert: Luftverkehrsmanagement, Produktionsautomatisierung, und intelligente Transportsysteme. Die Evaluation basierte auf Prototypenimplementierungen, Studien, Benchmarks und theoretische Beweise. Die Resultate der Evaluation zeigen eine höhere Koordinationseffizienz, bessere Robustheit gegenüber sich ändernden Geschäftsanforderungen, einfache Umsetzung von Geschäftsanforderungen und eine Reduzierung der Komplexität in Applikationskomponenten.

Table of Contents

1. Introduction ... 1
   - 1.1. The Interaction Problem ... 2
   - 1.2. The Agility Problem ... 3
   - 1.3. Contribution of this Thesis ... 5
   - 1.4. Challenges Related to this Thesis ... 6
     - 1.4.1. Pervasive and Trustworthy Network and Services Infrastructure ... 7
     - 1.4.2. Cognitive Systems, Robotics and Interaction ... 8
     - 1.4.3. ICT for Mobility, Environmental Sustainability and Energy Efficiency ... 8
   - 1.5. Organization of this Thesis ... 9
2. Related Work and Background ... 11
   - 2.1. Complex Systems ... 11
     - 2.1.1. Terms and Definitions ... 11
     - 2.1.2. Characteristics of Complex Systems ... 11
     - 2.1.3. Types of Complex Systems ... 12
     - 2.1.4. Methods of Managing Complex Systems ... 15
     - 2.1.5. Research Experts' Experiences ... 17
   - 2.2. Coordination Theory ... 19
     - 2.2.1. Definition ... 19
     - 2.2.2. Coordination Models ... 21
     - 2.2.3. Coordination Frameworks ... 24
   - 2.3. Software Architecture ... 35
     - 2.3.1. Software Architecture Definition ... 36
     - 2.3.2. Software Architecture Concepts and Principles ... 37
     - 2.3.3. Software Architectural Styles ... 39
   - 2.4. Software Evolution ... 46
     - 2.4.1. Motivation for Change ... 46

2.4.2. Adapting Architectural Elements ... 47
2.4.3. Architectural Tactics supporting Adaptations ... 48
3. Research Contribution ... 52
3.1. Research Issues ... 53
3.1.1. Interactions in Complex Software Systems ... 54
3.1.2. Evolution of Complex Software Systems ... 55
3.2. Research Methods and Evaluation Concepts ... 56
3.2.1. Research Methods ... 57
3.2.2. Evaluation Concept ... 57
3.3. Application Scenarios ... 57
3.3.1. Real-time, Safety-related Traffic Telematics (RealSafe) ... 58
3.3.2. System-wide Information Sharing (SWIS) ... 61
3.3.3. Simulation of Assembly Workshops (SAW) ... 63
3.3.4. Summary ... 65
4. The Space-based Computing Paradigm ... 67
4.1. SBC Overview ... 67
4.1.1. Coordination Policies ... 68
4.1.2. Profiles Enabling Agility ... 69
4.2. XVSM- eXtensible Virtual Shared Memory Architecture ... 70
4.2.1. Container-Engine ... 71
4.2.2. XVSM Runtime ... 75
4.2.3. XVSM-Application API ... 79
4.2.4. Supported Ways of Decoupling ... 83
4.3. Mapping Architectural Styles ... 86
4.3.1. Data-centric Architectural Styles ... 86
4.3.2. Dataflow Architectural Styles ... 89
4.3.3. Explicit Invocation Architectural Styles ... 90
4.3.4. Implicit Invocation Architectural Styles ... 92
4.4. Discussion ... 93
4.4.1. Interaction ... 94
4.4.2. Agility ... 95

5. Prototypic Realization of the Application Scenarios 98
5.1. RealSafe 98
5.1.1. Requirements concerning the Architecture 98
5.1.2. Proposed Architecture 99
5.1.3. Limitations of Related Technologies 100
5.1.4. Description of the proposed Architecture 105
5.1.5. Summary 110
5.2. SWIS 110
5.2.1. Requirements concerning the Architecture 111
5.2.2. Limitations of Related Technologies 112
5.2.3. Description of the proposed Architecture 113
5.2.4. Summary 123
5.3. SAW 124
5.3.1. Requirements concerning the Architecture 124
5.3.2. Limitations of Related Technologies 125
5.3.3. Description of the proposed Architecture 126
5.3.4. Summary 129
6. Evaluation and Discussion 132
6.1. Evaluation of Application Scenarios 132
6.1.1. RealSafe Application Scenario 132
6.1.2. SWIS Application Scenario 143
6.1.3. SAW Application Scenario 147
6.2. Studies 157
6.3. General Discussion 160
6.3.1. Interactions in Complex Software Systems 160
6.3.2. Evolution of Complex Software Systems 162
7. Conclusion and Perspectives 165

List of Figures

Figure 1: Coupling between application components and dependency between application component and deployed architectural style ... 2
Figure 2: Control-driven Coordination [178] ... 22
Figure 3: Data-driven Coordination ... 23
Figure 4: Architectural styles and their categorization ... 39
Figure 5: Pipes and filter architectural style ... 40
Figure 6: Message-queuing as a dataflow architectural style ... 41
Figure 7: Example of a repository based architectural style [11] ... 41
Figure 8: Example of a blackboard based architectural style ... 42
Figure 9: Client/Server architectural style ... 43
Figure 10: An example of a C2 architecture with four components in three layers, and two connectors that delimit the layers ... 45
Figure 11: Overview of the research challenges ... 52
Figure 12: Overview of research issues ... 53
Figure 13: The structure of the RealSafe V2I System [125] ... 59
Figure 14: Distribution of a set of Road Site Units in a road-network and meshed communication network [125] ... 60
Figure 15: The complex world of the Air Traffic Management domain [156] ... 61
Figure 16: Heterogeneous network infrastructures and coordination requirements [153] .. 63
Figure 17: View of a simulated Production Automation System [124, 226, 227] ... 65
Figure 18: High-level view of the Space-Based Computing Paradigm ... 67
Figure 19: Examples for client/server (left) and distributed architectures (right) for a space ... 68
Figure 20: Overview of the Space-Based Computing Architectural Style ... 69
Figure 21: XVSM and its various implementations ... 70
Figure 22: XVSM architecture with a container hosting a random-, a FIFO-, and a PRIO coordinator structuring 7 entries [124] ... 71

Figure 23: Execution sequence and return values of Aspects and data- and control-flow in a container with three installed pre- and post-Aspects [125] 76
Figure 24: The concept of a Virtual Container 77
Figure 25: Architecture of the XVSM Runtime Layer 78
Figure 26: The concept of Answer Containers 80
Figure 27: General structure of an XVSM Notification [127] 81
Figure 28: The five categories of decoupling in XVSM [151] 83
Figure 29: Repository architectural style realized with XVSM concepts 87
Figure 30: Replicated repository architectural style using database specific strategies for consistency management 88
Figure 31: Replicated repository architectural style using pre-aspects for consistency 88
Figure 32: Batch sequential architectural style realized with XVSM concepts 89
Figure 33: Pipe and filter architectural style realized with XVSM concepts 89
Figure 34: Message-Queuing architectural style realized with XVSM concepts 90
Figure 35: The client/server architectural style realized with two containers 91
Figure 36: The client/server architectural style with two coordinators 91
Figure 37: Implicit invocation architectural style 92
Figure 38: Implicit invocation architectural style in larger networks 93
Figure 39: Relation between road network – road segments - RSU 99
Figure 40: A centralized architecture approach 101
Figure 41: Handling container replicas [123] 107
Figure 42: The operation of SBC and DHT concepts in a publish/subscribe scenario [125] 109
Figure 43: Configuration of the SWIS platform 115
Figure 44: Components of a SWIS Node 118
Figure 45: Processing of messages in a SWIS node 119
Figure 46: Components of a Shadow Node 121
Figure 47: Distribution category of SWIS Node 1 122
Figure 48: Reaching group decisions in the distribution category of XVSM 123
Figure 49: Data structures for storing routing tables 127
Figure 50: Production automation system split into several DHT lookup areas 128
Figure 51: Production agents with triple replicated containers using various DHT lookup areas (replication clusters) 129

Figure 52: Complexity comparison in case of retrieving [123] 133
Figure 53: Time spent in the system versus message entrance time in ms [19]................ 137
Figure 54: Total message throughput [19] 138
Figure 55: The development of the size of a message queue in case data cannot be retrieved sufficiently by vehicles within the connection window 141
Figure 56: Communication between components using queues and containers with a simple transformation instruction 144
Figure 57: Communication between components using queues and containers with an aggregating transformation instruction 146
Figure 58: Communication between components using queues and containers in case of Shadow Nodes 147
Figure 59: Initialization time for a single Crossing Agent 149
Figure 60: Initialization time for multiple crossing agent 150
Figure 61: Comparing the complexity of prioritized queues with the container concept (P..entry) 153
Figure 62: Comparing the complexity of a prioritized queue of the traditional Linda approach with the container concept (P..entry) 155

List of Tables

Table 1: Coordination frameworks and its coordination capabilities in comparison to original Linda 35

Table 2: Costs and routes from every container to destination DS1 126

Table 3: Durations [ms] for the retrieval of a message out of 10, 100 and 1000 entries .. 135

Table 4: Durations [ms] for the retrieval of 10, 100 and 1000 entries [123]........ 142

Table 5: Comparison of crossing recovery; message based vs. replicated Space Containers 151

Table 6: Comparison of multiple crossing recovery; message based vs. replicated containers........ 152

Table 7: Time in ms to retrieve a single entry using different coordinators [124]........ 156

Table 9: Reported lines-of-code and effort for comparison of CORSO and RMI with MozartSpaces 160

List of Listings

Listing 1: Retrieving region-specific information with a well-known key pair. 134
Listing 2: Retrieving region-specific information with a well-known key pair. 134
Listing 3: Retrieving a FIFO sorted entry with Linda. 156

Glossary

ACG	Austro Control
AOP	Aspect-oriented Programming
API	Application Programming Interface
ASD	Agile Software Development
DHT	Distributed Hash Table
ESB	Enterprise Service Bus
ITS	Intelligent Transportation System
JMS	Java Message Service
MAS	Multi-agent System
MDA	Model-driven architecture
MDSC	Model-driven system configuration
RealSafe	Real-time Safety-related Traffic Telematics
RSU	Road Site Unit
SBC	Space-Based Computing
SAW	Simulation of an Assembly Workshop
SWIS	System-wide Information sharing
TCC	Traffic Control Centre
XVSM	eXtensible Virtual Shared Memory

Chapter 1

1. Introduction

Complex systems are systems [206] whose properties are not fully explained by an understanding of their single component parts. Complex systems (e.g., financial markets, bacteria life cycles) usually consist of a large number of mutually interacting, dynamically interwoven, and indeterminably dis- and reappearing component parts. The understanding often is, that the complexity of a system emerges by interaction of a (large) number of component parts, but cannot be explained be looking at the parts alone.

The subject of this thesis relates to software systems that can be interpreted as complex systems as well. Such systems, especially software-intensive systems [29], usually interact with other software, systems, devices, sensors and people. Examples include large-scale heterogeneous systems, systems for avionic applications, or telecommunication. Over time these systems become more distributed, heterogeneous, decentralized and interdependent, and are operating more often in dynamic and frequently unpredictable environments. Therefore, software developers have to deal with issues like heterogeneity and varying size of components, variety of protocols for interaction with internal and external components, number of potential incidents, like crashed or unreachable components in distributed environments, or adaptability of the system throughout its lifetime.

In the course of developing distributed software systems, software developers cannot avoid coping with the aforementioned complexity issues. They have to deal with higher-level architectural concepts (section 2.3) as well as the transformation of the design of the given software architecture to actual system implementations, system tests, and system evaluations. The complexity of a software system can be categorized according to aspects [8] like task structure, unpredictability, size, algorithmic, and its chaotic characteristics. Concepts and mechanisms like abstraction, decoupling, simplicity, layering, or the documentation of already gained experiences (see section 2.1.4) allow software developers to deal with these complexity issues in software systems. Documentation in this context may contain experience reports, recommendations, and lessons learned represented by means of e.g., software design patterns or architectural styles (see section 2.3). Today's software systems typically consist of mainly distributed application components representing higher-level business goals and a middleware technology usually representing an architectural style and abstracting the complexity concerns related to network and distribution. However, software developers still have to

deal with the interaction of application components and with the agility of the architecture due to changing business requirements.

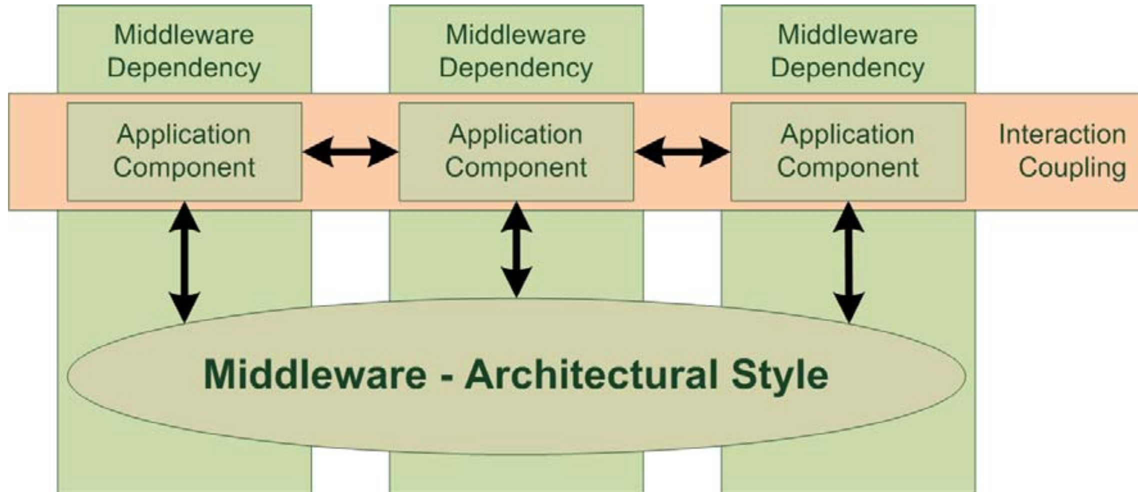

Figure 1: Coupling between application components and dependency between application component and deployed architectural style

1.1. *The Interaction Problem*

The message-passing paradigm is a common concept allowing application components to communicate with each other. The message-oriented middleware [101, 150], prominent representative is the Enterprise Service Bus (ESB) [40], provides synchronous and asynchronous message-passing properties and promises to interconnect application components in a loosely coupled manner. Since message-oriented middleware is only capable of transmitting and transforming messages between application components, it lacks support for complex interaction requirements which involve the participation of several application components for decision making, like in the telecommunication domain [112]. The software developers have no other choice but to take into account both, the application logic representing the business goal and additional coordination logic needed to fulfill the specific coordination requirement. Such logic for instance may contain implementation matters related to synchronization problems. Furthermore, it may include logic for the management and supervision of the latest state of the coordination process itself otherwise the application may get "lost" and the common business goal cannot be reached. Additional management is needed in case the coordinating component crashes and after recovery the failed application component still wants to be part of the running coordinating process. These additional issues introduce potential sources of error, decrease the efficiency of the system, and increase the cognitive complexity [10, 144] of the application component. However, the responsibility of the software developer should focus on the application's business goals and not on concerns related to distribution or coordination.

A framework that has been explicitly designed for coordination purposes is the so called tuple space, based on the Linda coordination model of David Gelernter [78]. It is a data-centered, blackboard based, architectural style that describes the usage of a logically shared memory, the tuple space, by means of simple operations as interaction mechanisms. The approach promotes a clear separation between the computation model, used to express the computational requirements of an algorithm, and the coordination model, used to express the communication and synchronization requirements. The state of coordination is not embedded in the coordinating process itself but in the space [177]. The state of the coordination information on the blackboard determines the way of execution of the process. By means of this coordination model the application may entirely focus on business goals since the model "*gives application builders the advantage of ignoring some of the harder aspects of multi-client synchronization, such as tracking names (and addresses) of all active clients, communication line status, and conversation status*" [130]. The Linda coordination model uses template matching with random, non-deterministic tuple access to coordinate processes (see sections 2.2.2.2, 2.2.3) and is limited to fixed Linda coordination policies due to its static coordination model. This limitation restricts the benefit of using such a communication abstraction. Therefore, with respect to more complex coordination requirements the software developer still needs to implement coordination functionality not directly supported by the coordination framework within application components. Consequently, this increases the complexity of the application component, decreases performance, and leads once again to an unclear separation between computation model and coordination model.

Figure 1 depicts the interaction problem as *Interaction Coupling*. Coupling in this context is interpreted as the amount of coordination logic implemented in application components, representing a coordination goal. It is not used in the sense of coupling between application components based on their ways of communicating with each other (RPC vs. message-oriented).

1.2. *The Agility Problem*

Business is in constant change and in case of software systems both the software development process and the underlying software architecture has to cope with and satisfy such unpredictable requests for adaptation. The field of Agile Software Development (ASD) [47, 61, 66, 95] addresses exactly the challenges of an unpredictable, turbulent business and technology environment and provides capabilities to handle agile requirements. The concepts of ASD have been created by experienced practitioners and can be seen as a reaction to e.g., plan-based methods, which attach value to "a rationalized, engineering-based approach" [161]. There, problems are seen to be fully specifiable and solvable with an optimal and predictable solution. It is believed

that by means of extensive planning and codified processes, development can be made efficient and predictable. In contrast, agile software development has been proposed as a solution to problems resulting from an unpredictable world. Several agile methods have evolved over time, like Dynamic Systems Development Method (DSDM) [207], Extreme Programming (XP) [18], Evolutionary Project Management (EVO) [84], or Scrum [198]. However, there is also skepticism [61] regarding agile software development with respect to architecture design and implementation issues. One is that agile development is an excuse for developers to implement as they like, coding away without proper planning and design [89, 188] and consequently causing suboptimal design-decisions [143, 210] and leading to the higher risk of introducing more and more architecture breakers. Furthermore, managing changing requirements is not done during the first project intending to create the first version of a product only, but it is an ongoing task throughout the entire product life [106].

The selection of a proper architectural style and thus the deployment of an appropriate middleware technology representing the chosen architectural style (see section 2.3) is an important and therefore difficult design decision. First, even if a specific problem can be solved by different architectural styles, software developers should select a style that matches the requirements of the existing problem best. The difficulty is that different architectural styles lead to a different design of a software system, as well as to designs with significantly different non-functional properties [200]. This means that the impact on e.g., performance, reliability, or security may be significant in case the wrong architectural style has been chosen for the given problem in the given context. The second aspect of decision making is the mutual influence of architecture and middleware on each other [56, 63]. Although architecture and middleware address different phases of software development, the predefined selection of a middleware technology automatically implies the usage and deployment of its components, which has an impact on the architecture of the system being developed. Conversely, specific architectural choices limit the set of usable middleware technologies in the implementation phase of the software development. In Figure 1 these two aspects are referred to as *Middleware Dependency*. Beside these two aspects, the process of decision making is influenced by business strategic or technical aspects pre-specifying the architectural style to be used by software developers or by software engineering aspects like how much experience do software developers have regarding the selected architectural style and how easy it is to learn it.

The introduction of changes may be triggered by new business requirements or by changes in the infrastructure. The former refers to business agility [109] while the latter to requirements like "the need for re-evaluation of the current software system solution due to more powerful network infrastructures". With respect to changing circumstances (i.e. the evolution of the software system) in complex systems, the two aspects of decision making may imply that in case of using insufficient concepts for the software architecture of the distributed system, new requirements may not be compatible with the

currently chosen architecture. A previously “clean” architecture has been “broken” by extending it with complex interconnections of concepts in order to satisfy the new requirement. In this thesis, requirements of this kind are also referred to as an “architecture breaker” that triggers significant design and code changes [202].

The complex interconnection of architectural concepts may be the result of two aspects. On the one hand, software architects practically use the concepts of the current architectural style to realize the new business requirement even if their capabilities do not entirely match that requirement. As mentioned in the previous paragraph, the properties (like performance or scalability) of the system may suffer. On the other hand, the integration of other, additional architectural styles into the software architecture to cope with and to be prepared for future requirements introduces additional cognitive complexity. For instance, frameworks introduce additional cognitive complexity by providing various features packed in a large, monolithic product [144]. This forces the software engineer to learn and to be aware of a large number of product concepts (before the product itself can be deployed). Consequently, instead of a stable and easily manageable set of architectural concepts for effectively managing complexity concerns, the number of concepts a software developer has to know and work with is increasing with the size and degree of evolution of the system. Either way, the dilemma is that an unnecessarily complex or even dysfunctional system structure may emerge [56]. Software developers are interested in architectural frameworks that abstract system complexities from application components with the result that the number of architectural concepts software developers use stay stable even if the system evolves.

1.3. *Contribution of this Thesis*

Taking into account the previously mentioned issues regarding the interaction and agility problem in software systems, this thesis proposes the so called Space-Based Computing (SBC) paradigm to support and facilitate software developers efficiently in their efforts to control these complexity concerns in software systems. We define the SBC paradigm as an architectural style that combines and abstracts the properties of several different architectural styles and extends them by sophisticated coordination models.

Regarding the interaction problem, SBC extends and strengthens the clear separation between computation and coordination logic by allowing the selection and injection of scenario specific coordination models. From the application’s point of view the SBC architectural style is comparable to the blackboard architectural style (see section 2.3.3.2), orientated on the Linda coordination language. In contrast to traditional Linda coordination frameworks, the SBC architectural style extends the Linda coordination model by introducing exchangeable mechanisms for structuring giving the data in the space special ordering characteristics and reducing dependencies between application

components and coordination models. SBC explicitly embeds sophisticated coordination capabilities in the architectural style, and thus makes the style itself dynamic with respect to the scenario's coordination problem statement. This means that SBC is capable of abstracting coordination requirements and changes from the application. Since coordination requires and thus inherently consists of communication, consequently the abstraction of coordination also means that SBC abstracts communication requirements as well.

With respect to the agility problem, SBC defuses "architecture breakers" by abstracting problematic differences between traditional architectural styles. The strength of the style is facilitated by combining and abstracting the characteristics and properties of several different architectural styles captured in a simple API and thus minimizing dependencies between application components and architectural styles.

Since SBC is an architectural style it needs to be realized, implemented and evaluated in different domains to prove its feasibility and generality. The SBC architectural style has been realized in the reference architecture called XVSM[1] (eXtensible Virtual Shared Memory) as a proof-of-concept. The architectural design of XVSM has been further materialized in the Java implementation called MozartSpaces[2] and the .Net implementation named XCoSpaces[3]. This thesis bases on founded research projects lasting several years with the industrial application domains used for evaluation purposes referred are: intelligent transportation systems, distributed services in air traffic management, and production automation systems. With respect to these domains the research issues focus on aspects like feasibility, effort, robustness, performance, complexity, and usability to evaluate the SBC architectural style.

Major results of the evaluation in this context are higher coordination efficiency accompanied with minimized complexity within application components, and improved robustness against software architecture changes as well as an efficient and simplified implementation of changing requirements with an invariant number of concepts used to realize the changes.

1.4. *Challenges Related to this Thesis*

The research contributions of this thesis and the three mentioned application domains are positioned within a bigger research context. The context is specified by the seventh EU framework program.

---

[1] www.xvsm.org

[2] www.mozartspaces.org

[3] www.xcoordination.org

The seventh EU framework program "*Cooperation Work Programme: Information and Communication Technologies*" (ICT)[4] lists several research challenges to be investigated in the next several years. The program's objective is to improve the competitiveness of the European industry and to strengthen Europe's scientific and technology base in order to ensure its global leadership in ICT. The program defines three future technologies and socio-economic transformations as research drivers to ensure sustainability in Europe's economy: "*Future Internet*", "*alternative paths to ICT components and systems*", and "*ICT for sustainable development*". Based on these three topics and covered by a set of research objectives, seven major challenges have been identified: three of them refer to technological challenges and the other four are driven by socio-economic goals.

In the following three of the seven challenges are briefly described and it is explained how the thesis and the selected use case domains relate to those challenges. The application scenarios are the Traffic and Transportation scenario (RealSafe), the Air Traffic Management scenario – System-wide Information Sharing (SWIS), and the Production Automation scenario - Simulation of an Assembly Workshop (SAW). The scenarios are described in section 3.3 while related work regarding the scenarios is presented in sections 5.1.3, 5.2.2, and 5.3.2.

1.4.1. Pervasive and Trustworthy Network and Services Infrastructure

The first technological challenge copes with "*pervasive and trustworthy network and services infrastructure*" to replace the current Internet, mobile, fixed and audiovisual networks. One of the objectives of this challenge is to enable "*dynamic, efficient and scalable support of a multiplicity of user requirements and of applications with various traffic patterns, variable end-to-end quality of service, point-to-point or point-to-multipoint distribution modes, and supporting legacy and future service architectures*". Beside efficient routing, location independent addressing or naming, and end-to-end content delivery techniques, the evolution and interoperability of services are research issues as well.

The SWIS (see section 3.3.2) project deals with an information-sharing network within the Air Traffic Management domain with extremely demanding safety and security requirements, as well as the need for high availability. There is the need to cover various coordination requirements among services and between individual network nodes, abstract the heterogeneity of the used network infrastructures, and to support an effective adaptation of the SWIS platform due to changing business requirements. Although the domain is restricted to closed, non-public networks rather than Internet

[4] ftp://ftp.cordis.europa.eu/pub/fp7/ict/docs/ict-wp-2009-10_en.pdf

infrastructures, the project demonstrates the communication behavior of heterogeneous services using heterogeneous network infrastructures in a geographically wide scale scenario. This project was carried out in cooperation with Frequentis AG, Austro Control (ACG), and the Institute of Information & Software Engineering Group (IFS) at the Vienna University of Technology.

1.4.2. Cognitive Systems, Robotics and Interaction

The second technological challenge is about "*cognitive systems, robotics and interaction*" to enable the engineering of context-aware and easy-to-use ICT systems. The objective of this challenge is to engineer systems that are capable of sensing and understanding unstructured environments without the need of specifying appropriate workflows and concrete reactions to every eventuality the system will have to cope with during its execution. Those systems should autonomously or in cooperation with people execute tasks that were not planned in detail at design time. This implies that among others the system should be capable of adapting itself to changing requirements with minimal human intervention as well as coping with unpredictable events without decrease of performance and effectiveness due to anticipation of events at some point in the future.

The SAW (see section 3.3.3) project aims to improve the effectiveness of production automation systems by improving the coordination and communication capabilities of software agents. Software agents represent functional machine parts and are critical with respect to the correct functionality of the entire system. In this environment an unpredictable event is e.g., the sudden failure of such an agent the system needs to cope with by facilitating continuous coordination capabilities. This project was carried out in cooperation with Rockwell Automation, Automation and Control Institute (ACIN), and the Institute of Information & Software Engineering Group (IFS) at the Vienna University of Technology.

1.4.3. ICT for Mobility, Environmental Sustainability and Energy Efficiency

The third technological challenge takes ICT into account with respect to "m*obility, environmental sustainability and energy efficiency*". The aim of this challenge is "*to provide new intelligent systems that assist the driver to avoid accidents, provide drivers with real time information to avoid congestion, and optimize a journey*". Beside autonomous on-board systems and vehicle-to-vehicle technologies, the focus also lies on vehicle-to-infrastructure co-operative technologies and flexible traffic

network management. All in all, the main objective is to improve road safety, to increase performance and to reduce costs of the technology.

The RealSafe project (see section 3.3.1) operates within the domain of Intelligent Transportation System (ITS) and aims to provide a scalable and decentralized vehicle-to-infrastructure technology in order to improve driver-safety by offering information relevant for the decision making process. The project requires abstractions related to complexity issues of distributed systems, like replication strategies and fault-tolerance mechanisms, for the developer and the possibility of a robust, effective and efficient time-constrained coordination of stakeholders and exchange of safety-critical information between the vehicle driver and the infrastructure. This project was carried out in cooperation with Kapsch TrafficCom AG, Autobahnen- und Schnellstraßen-Finanzierungs- Aktiengesellschaft (ASFiNAG), the Telecommunications Research Center Vienna (FTW), nast consulting, and Institute of Communications and Radio-Frequency Engineering (INTHFT) at the Vienna University of Technology.

1.5. *Organization of this Thesis*

This thesis is structured as follows: Chapter 2 summarizes related work on complex systems, coordination platforms, software architectural styles, and issues of software evolution. Chapter 3 explains the research approach by presenting the research contributions, describing the research methods and introducing the application scenarios. Chapter 4 describes the concepts of the Space-based Computing paradigm and the general reference architecture of the eXtensible Virtual Shared Memory (XVSM) implementation. Chapter 5, concentrates on the description of the prototype architecture of the application scenarios, and chapter 6 presents the evaluation results with regard to the specified research issues. Finally, chapter 7 concludes the thesis and provides further research issues for future work.

Chapter 2

2. Related Work and Background

Based on the problem statements given in chapter 1, this chapter summarizes related work on relevant technologies with emphasis on complex systems, coordination theory, software architecture, and software evolution.

2.1. *Complex Systems*

This section motivates management of complexity and complex systems, and is mainly based on the research findings of a systematic literature review [22]. As first step various definitions of complexity and characteristics of complex systems are listed. Additionally, examples for different types of complex systems are given. Finally, methods for managing complexity are presented.

2.1.1. Terms and Definitions

The literal definition identifies complexity as the "*quality or state of being complex*". Complex moreover is defined as "*a whole made up of complicated or interrelated parts, composed by two or more parts, which are hard to separate, analyze or solve, and which may be concerned with or may contain complex numbers*" [169, 199]. Additionally, a complex system is understood as "*a system whose properties are not fully explained by an understanding of its component parts*". Complex systems usually consist of a large number of mutually interacting and interwoven parts, entities or agents [46]. Finally, the engineering of complex computer systems can be seen as "*all activities pertinent to specifying, designing, prototyping, building, testing, operating, maintaining, and evolving a complex computer system*" [7].

2.1.2. Characteristics of Complex Systems

Typically, complex systems share some basic characteristics [144, 174] and therefore systems are often called "complex systems", if

a. they contain a large number of potentially interacting components the system consists of ("connectivity or structural complexity")

b. the number of possible states or the relation of the number of possible and suitable solutions of a complex problem is high
c. the system consists of heterogeneous system entities (“diversity”)
d. linear modeling is not powerful enough to model and to understand the particular system
e. advanced dynamic approaches (e.g., simulation) are needed to understand the particular system.

Another possible separation of complex systems is the differentiation of essential and accidental complexity of problems and solutions [28]. Essential complexity cannot be reduced but only shifted. On the other hand accidental complexity is not essential and therefore can be reduced, e.g., by a better representation/model of the problem.

2.1.3. Types of Complex Systems

Concrete examples for the different types of complex systems can be found in nearly all sciences. First, an exemplary set of complex system types are briefly described to illustrate the diversity of complexity scenarios from domains outside and within computer science. Then, two examples of complex problem statements are given to show technical challenges, relevance for society, and needed approaches for coping with complexity.

Social Sciences

Social sciences try to model and simulate our society by discussing the interplay of elements of the system and its consequences, the dynamics of social process, the influence of social change and interdependence of microsystem and macrosystem to get hints for future behavior of societies. Examples from the literature are the simulation of the influence of political opinion leaders on voters using multiple information sources (like communication networks and self-selected news media), the analysis of the attitude of users to the use of information and communication technologies to determine its impact on learning of sciences, or the comparison of a model of ethnic mobilization with a model of hierarchy declination [3, 134, 162, 175].

Economic Sciences

In economics, the simulation of economic processes or phenomena is one of the major goals. However, the number of variables or parameters often does not allow a discrete simulation, making the overall goal a very complex one. Typical problems include the modeling of consumer behavior and fluctuations in economic activity, the determination of the effects of productivity on growth, the analysis of the causes of underdevelopment,

or the modeling of financial markets with the final goal of a satisfactory determination of prices [82, 83, 100, 233].

Biology

In biology, the simulation of biological systems helps understand and predict the outcome of experiments to serve as a road map for the best way to proceed, but may never fully replace actual experimentation. Examples for biological complex systems are the simulation of bacteria life cycles and reproductions, the study of associations between infections and atherosclerosis under varied assumptions, the modeling of human glucose regulations depicted as nonlinear (logistic) function, or the analysis of genetic complexities. In genetics, one of the hardest complexity factors is the number of possible combinations or solutions, making tasks like identifying the role of infectious agents as the cause of diseases, the validation of simulated results based on theoretical formulations with respect to available clinical data, or the research on genome sequences very complex [87, 149, 160, 180, 211, 226].

Physics

In physics, the proof of complex theories like the Chaos Theory is one of the key research goals. Additionally, the properties of dynamic systems and the evolution of diverse systems, as well as the roots of the scientific concepts such as randomness, autocatalysis, nonlinear growth, information, patterns, etc are investigated [88].

To sum up, the major complexity criteria in domains outside computer science are the huge number of possible combinations/solutions, the impossibility to discretely predict system behavior due to the heterogeneity and complexity of single system participants/agents, and basic mathematical complexities like e.g., nonlinear growth. Possible solution approaches are computing power, system modeling, and simulation of complex systems.

Computer Science

In business informatics the main aspects of complexity as a whole are the vast number of different actors/stakeholders and their interests and abilities: organizations like insurance companies and health service providers on a national level, and stakeholders like doctors, nurses and patients on the level of a basic organizational unit [111].

In software engineering the complexity of software [218] is thought to primarily lie in the wealth of usage context, e.g. understandability for the user and usability in certain real-life non-ideal circumstances. One big source of complexity is the consideration of end-users' experiences with computers in creating design environments with easy to explore and easy to grasp concepts that help software developers satisfy end-users' cognitive needs and deal with contextual issues (like aesthetic, practical, and social

properties of the application). Additionally, end-user – computer interaction opens up a huge space of possible scenarios, which is also seen as a source of complexity.

For some authors from the fields of computational intelligence, complexity itself is simply the unknown laws and rules [28] of interaction between biomolecules taking place inside human head that have to be discovered, and the incorporation of newly learned concepts. An example is the comparison of the current possible level of computer intelligence to that of children of various ages.

In the domain of formal methods, the authors in [25] define complexity as the amount of possible unknown behavior of a given design. According to the definition the difficulties arise from not knowing enough about the influences of anything onto a design, or a part thereof. The authors claim to be dealing more with the avoidable part of complexity in designs, but also acknowledge the existence of a minimum “irreducible complexity” within designs, given by its design goals.

Another example is fractal forms [129]. They are complex because they are infinitely detailed, with (some of) their parts self-similar but not necessarily identical copies. They can be produced by simple generator functions, but getting back to the generator functions from the created fractal is usually impossible. Complexity can be measured by means of (multi-)fractal analysis which can produce certain high-level descriptions of fractal system, e.g. the fractal dimension.

In the domain of Media and Vision, imperfect information and competing goals of each actor in a multi-person design process is another source of complexity [142]. Examples are e.g. the domain of commercial airplanes and car industry. There, safety is a complex [159] socio-technical problem with no simple solution. The technical flaws that lead to accidents can often be traced back to root causes in the organizational culture. Thus, concentrating exclusively on technical issues and ignoring managerial and organizational deficiencies will not result in effective programs for increasing safety. Dealing with such problems will require experts in multiple fields, such as system engineering, software engineering, cognitive psychology, and organizational sociology, working together as a team.

In the field of distributed systems the Large Hadron Collider (LHC) [75] is another representative source of complexity. The main aspect of complexity is the sheer amount of data to be distributed and stored in a fault-tolerant way, and the processing power needed to deal with it. Another example is the Internet itself consisting of a large set of heterogeneous networks, services, protocols, or data, which are changing regularly.

In technical informatics several practical complex problems can be found. This includes traffic-related topics like the simulation of road traffic or the self-organization of traffic lights in order to improve traffic flows, as well as topics from production control, like the functioning of manufacturing cells, in which several complex systems have been joined, or the monitoring and optimization of industrial production processes [17, 26, 80, 212].

Cross-domain scenarios

In the following two example [65, 98] for a cross-domain complex system is illustrated to show technical challenges and relevance for society.

An example is given with respect to the search for sustainable future energy sources since for future economic wealth and further social development it is important to have a trustworthy, reliable, and affordable power supply. The aim is to reduce energy consumption and to intelligently make use of current and forthcoming oil resources while increasing profitability of alternative energy sources like biomass, off-shore parks, or photovoltaic. The propositions are necessary due to high increase of CO_2 emissions and its obligatory control of intent of reduction. Furthermore, the integration of alternative and sustainable energy resources into current power supply grid is a challenging task due to e.g., fluctuations in the amount of power generated. Complexity challenges refer to the rapid collection of widely geographically distributed CO_2 monitoring data and its transmission, processing, and analysis. Complexity issues also refer to fast acquisition, transmission, and processing of data to produce near real-time control responses regarding power generators due to changes in the power grid. In the course of managing such systems algorithms and models are needed to predict power consumption.

Another example refers to recognition and controlling of spreading diseases. The problem is that inefficient and insufficient reactions may influence magnitude and speed of spreading significantly. However, a proper assessment of the current situation to find appropriate intervention techniques is difficult due to non-linear and dynamic effects. For instance, it has to be evaluated whether public transportation centers, like airports and railways stations have to be closed down in which area. The challenge of such a complex system is the simulation of individuals and their movements at all levels of society. Prediction models are necessary to help identifying the magnitude of spreading and the assessment of the effects of suitable countermeasures at local, national, and global level. So far, optimized models exist for each single level and simulations capturing 300 million individuals have been performed. However, simulations in this context need to investigate which countermeasures work best at which level and how levels mutually influence each other. Preconditions for such challenging tasks require hardware at supercomputing level, validations of real-time models, assessments of parameters based on historical data, and near-real-time updates of data collected about population.

2.1.4. Methods of Managing Complex Systems

The most common methods and mechanisms [4, 8, 25, 76, 144, 168] used by the various fields of computer science to control complexity are:

- Simplicity: managing complexity can be supported by the so called “KISS - Keep It Simple, Stupid” principle (e.g., divide and conquer), trying to dumb down the design as much as possible with respect to the given design goals. The drawback is the problem of reassembling the simple solutions.
- Abstraction: by means of generalization simplification of a scenario, the information content of a concept is reduced by focusing only on those properties that are relevant for the particular purpose and by omitting the irrelevant details.
- Decoupling: is used to identify the separation of system components that should not depend on each other.
- Decomposition: is the process of breaking down a complex system into various components that are easier to understand, manage, or maintain. It limits the designer’s scope by either dividing large problems into smaller, relatively isolated and manageable parts.
- Classification: a categorization of system parts with similar properties into groups to introduce some structure and recognize relationships between the individual categories.
- Standardization: the process of developing and agreeing upon a technical standard with the benefit of a structured and non-dynamic environment.
- Modeling: is the process of generating an abstract and simplified view of a complex system including representations of empirical objects or physical processes allowing simulations to run.
- Transformation: an approach to manage the given complexity is the transformation of the given problem to a domain with proven solution approach, and then the transformation back to the original domain.
- Debugging and Testing: in case there are no ready and tested solutions for a given complex problem, the solution for that problem statement has to be found in a testing process. The exploration of the range of possibilities and the usage of tools for debugging and testing are another essential parts of dealing with complexity.
- Simulation: is an attempt to study a modeled real-world behavior of a system by varying parameters. It allows conclusions about how the system works, and predictions may be derived.
- Experience: already gained experiences from experienced contributors documented in patterns, development processes, reports, lessons learned analyzes, or guidelines help inexperienced software developers to cope with complex systems faster. Therefore, software developers should be sufficiently qualified to be able to make the correct decisions for situations that are not covered by detailed guidelines. These guidelines, on the other hand, will cover

mainly standard situations and leave decisions in every unusual situation up to the qualified individual.

2.1.5. Research Experts' Experiences

For completion and supplementation of the conducted literature review we also performed interviews with researches in various fields of computer science. The interview sessions had the intention to collect information about aspects of complexity and methods for managing complexity with respect to their research fields and experience gained from real-world scenarios. Interview partners were Prof. Gerti Kappel, Prof. Karl-Michael Göschka, Prof. Jens Knoop, Prof. Peter Purgathofer, Prof. Peter Puschner, Prof. Gernot Salzer, and Dr. Alexander Schatten from the Vienna University of Technology covering research fields related to programming languages, compilers, verification, automatic parallelization and optimization, formal methods, computational logic, component-based software engineering, model engineering, web engineering, process engineering, software architectures, service-oriented computing, distributed systems, embedded systems, real-time systems, real-time programming languages, and user interface design.

Interview partners see complexity in

- the frequent changing of general conditions and parameters of markets and technologies,
- the increasing number of interconnection between more and more communicating entities resulting in exponential growth of data traffic,
- the scale and heterogeneity of components and the dynamically changing dependencies between those components when the system gains properties like emergent behavior or self-organization,
- the heterogeneity of knowledge, expectations, and interests of roles participating in the design of software systems and the increasing requirements of customers regarding the quality of produced systems,
- the ratio between possible and useful solutions for a given problem space and also in the process that finds useful solutions, and
- the decidability, i.e. how meaningful are assumptions and statements and how long does it take to prove them.

An interesting comment of one of the interview partners refers to a clear distinction between computer science and other natural sciences. The researcher mentions that in other natural sciences no new complexities may be "created" due to natural boundaries.

The results of the interviews revealed that computer science copes with complexity issues at three different points:

- complexity with respect to the work with stakeholders, their ambiguous views on a real life problem, and the mapping of those into concepts of computer science,
- complexity that is related to the interactions and interrelations of system components derived from models, and
- the inherent complexity of these components.

The following list presents methods mainly used by the researchers to manage complexities within their fields:

- Principle of abstraction and decomposition (separation of concerns)
- Classification, partitioning, and segmentation
- Open standards for interaction between components
- Agile management, adaptivity (e.g., switching between protocols), or concepts of long-term evolution (e.g., reprogramming during runtime) but also structuring of elements in hierarchies or by prioritization
- Support of creating fault-tolerant systems both by means of concepts during runtime and by means of tools during design time
- Monitoring to get an insight into the system
- (domain-specific) Modeling and simulations
- "Moodling through", i.e. making good local decisions and then hoping for the best
- Formalization and definition of problems correctly taking into account the problem statements
- Simulations to evaluate a range of parameters

The conducted interview sessions confirmed the results of the literature review. Most of the mentioned methods for managing complexity are defined in literature as well. Personal experience with complexity (like "moodling through") plays an important aspect. An interesting result of the interviews is the controversial relation to simulations. While simulation is important in e.g., model engineering, researchers of formal methods explained that it does not help at all improving understanding and therefore it is not used either.

2.2. *Coordination Theory*

Since significant characteristics of complex systems refer to the interaction between components of complex systems, coordination between these components is an important issue to be investigated. This section summarizes related work on coordination theory by giving a definition of coordination, describing coordination models, and presenting technologies built for supporting coordination.

2.2.1. Definition

Coordination [138] is the additional organizing activity (like information processing) that is needed in case multiple actors pursue the same goal, that a single actor would not perform. In a more general perspective [139], coordination refers to "the act working together harmoniously". However, it can be derived that coordination itself consists of different components, like actors performing some activities which are directed to a goal. Therefore, the definition implies that activities are not independent and thus coordination can be seen as "*the act of managing interdependencies between activities performed to achieve a goal*". Later, Malone and Crowston [140], the founders of interdisciplinary science of coordination theory, describe their definition in a refined form just as "*managing dependencies between activities*". It has to be pointed out that coordination makes only sense if tasks are interdependent. If there are no interdependences, there is nothing to coordinate either. Beside Malone and Crowston's definition, [228] summarizes definitions of coordination:

- Coordination is structuring and facilitating transactions between interdependent components.
- Coordination consists of the protocols, tasks and decision-making mechanisms designed to achieve concerted actions between interdependent units.
- Coordination describes the integrative devices for interconnecting differentiated sub-units.
- Coordination consists of the joint efforts of independent communicating actors towards mutually defined goals.
- Coordination refers to networks of human action and commitments that are enabled by computer communications technologies.
- Coordination composes purposeful actions into larger purposeful wholes.
- Coordination describes actions and decisions of individual actors within an organization which need to be timely attuned for the organization as a whole to realize its aim.

- Coordination is the integration and harmonious adjustment of individual work efforts towards the accomplishment of a larger goal.
- Coordination establishes attunement between tasks with the purpose of accomplishing that the execution of separate tasks is timely, in the right order and of the right quantity.

Given the unavoidable existence of dependencies, a characterization of different sorts of dependencies can be derived [140, 228]: shared resources, producer/consumer relationships, and simultaneity constraints.

Shared Resources

Every time multiple activities share a limited resource (e.g., money, data storage space) a process (e.g., mutual exclusion algorithms) is needed that manages resource allocation and thus the interdependency between those processes. A special case of resource allocation is task assignment, where a task is assigned to actors allocating the time they need to perform the task.

Producer/Consumer Relationships

The producer/consumer dependency relationship refers to the activity where a process produces something which is used by another process (the consumer). The relationship leads to several more dependencies:

- In the *prerequisite constraints* indicates that the consumer activity can only be started when the producer activity has finished. This requires at least a kind of notification process indicating to the consumer process that it can begin. Managing prerequisite dependencies also often involves *explicit sequencing* and *tracking* processes to make sure that producer activities have been completed.
- The produced object needs to be transported to the consumer, resulting in the physical transportation dependency or in case of information in communication dependency. Physical transportation often requires the storage of the object to be transferred. A special managing aspect *is just-in-time-delivery* where no storage is required. In parallel processing systems the rate of transportation must be regulated to make sure that the producer does not overwhelm the consumer.
- Usability is another dependency referring to the fact that the produced object is usable by the consumer. Standardization is a common approach to manage the dependency.

Simultaneity Constraints

The *simultaneity constraint* describes the dependency between activities which need to occur at the same time. *Synchronization* or *scheduling* of processes are methods for managing such dependencies.

2.2.2. Coordination Models

A coordination model [43] is either a formal or a conceptual framework to model the space of interaction. A formal framework expresses notations and rules for the formal characterization of coordinated systems, as used in frameworks [44] or [239]. A conceptual framework is required by software developers to manage inter-component interactions, since it provides abstraction mechanisms. In general, the emphasis is more on the expressiveness of the abstraction mechanism of the coordination model, and on its effectiveness helping software developers in managing interactions.

From a functionality point of view distributed systems are typically divided into the following three concerns:

- Computational logic (i.e. business logic) performs calculations representing the main intention of the system (i.e. business specific goals)
- Communication responsible for sending and receiving data from other components to be further processed.
- Coordination or dependency management responsible to execute tasks in a way where no dependencies are violated and the common coordination goal is achievable.

Sancese et. al. [194] argue that a clear separation of the three parts leads to a reduction of complexity of the entire systems also enabling a reliable and more stable implementation. The process of coordination follows a certain coordination model for which Ciancarini [43] defines a generic coordination model as a triple of {E, M, L}. In the model, {E} stands for either physical or logical entities to be coordinated. These can be software processes, threads, services, agents, or even human beings interacting with computer-based systems. {M} represents the coordination media (i.e. communication channels) serving as a connector between the entities and enables communication, which is a mandatory prerequisite for direct coordination [73, 228]. Such coordination media may be message-passing systems, pipes, tuple spaces [78] etc. {L} specifies the coordination laws between the entities defining how the interdependences have to be resolved and therefore, semantically define the coordination mechanisms. According to [177], existing variations of coordination models and languages can be mainly divided into two categories: control-driven (or task- or process-oriented) or data-driven coordination models, described in the following sections.

2.2.2.1. Control-driven Coordination

In control-driven coordination models [177] processes are treated as black boxes and any data manipulated within the process is of no concern to other system processes. Processes communicate with other processes by means of well defined interfaces, but it

is entirely up to the process when communication takes place. In case processes communicate, they send out control messages or events with the aim of letting other interested processes know their interest, in which state they are, or informing them of any state changes.

From a stylistically perspective, in the control-driven coordination model it is easy to separate the processes into two components, namely purely computational ones and purely coordination ones. The reason is that "*the state of the computation at any moment in time is defined in terms of only the coordinated patterns that the processes involved in some computation adhere to*" [177] and that the actual values of the data being manipulated by the processes are almost never involved enabling a coordination component written in a high-level language. Usually, a coordinator process is employed for executing the coordination code. The computations are regarded as black boxes with clearly defined input and output interfaces which are plugged into the coordination code, i.e. they are executed when the program reaches a certain part of the coordination code.

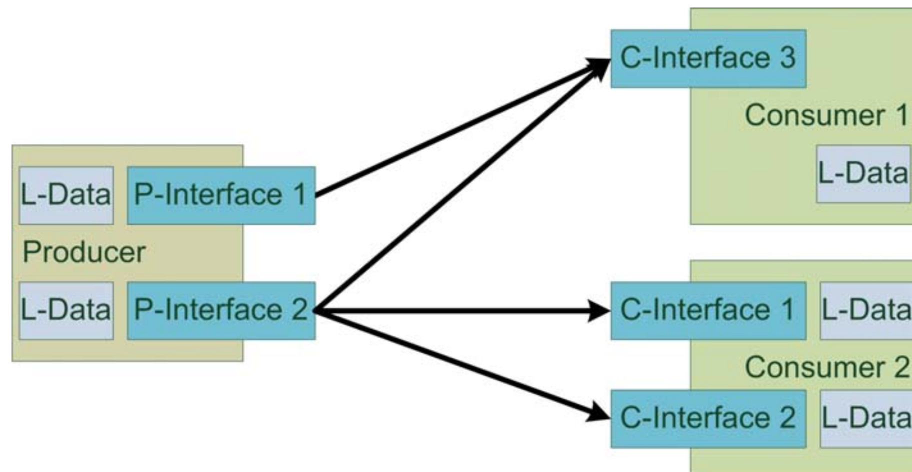

Figure 2: Control-driven Coordination [177]

Figure 2 is an example for a control-driven coordination models and shows a configuration consisting of producers with communication interfaces for sending events (P-Interface 1 & 2) and two consumers with communication interfaces for receiving events (C-Interface 1, 2, & 3). L-Data refers to variables stored in the process invisible to other components. The point is that communication takes place due to the semantics of the interface and data stored within the process itself is hidden to the outside environment. It is up to the producer to decide when changes are propagated. However, in which way (e.g., RPC [216], RMI [216], messaging [40, 101], publish/subscribe [54, 67, 103, 125, 221]) events are transmitted to the consumers is up to the middleware technology used in the given context. Examples for control-driven coordination languages include WS-BPEL [48], Manifold [9], CoLaS [53], or ORC [105].

2.2.2.2. Data-driven Coordination

In contrast to control-driven coordination models, the main characteristic of the data-driven coordination model is the fact that "*the state of the computation at any moment in time is defined in terms of both the values of the data being received or sent and the actual configuration of the coordinated components*" [177]. This means that a coordinated process is responsible for both examining and manipulating data as well as for coordinating either itself and/or other processes by invoking the coordination mechanism each language provides. A data-driven coordination language typically offers some coordination primitives which are mixed within the computational code implying that processes cannot easily be distinguished as either coordination or computational processes.

Carriero and Gelernter define in [79] that "*a coordination model is the glue that binds separate activities into an ensemble*". They express the need for a clear separation between the specification of the communication entities of a system and the specification of their interactions or dependencies; i.e. a clear separation between the computation model, used to express the computational requirements of an algorithm, and the coordination model, used to express the communication and synchronization requirements. They explain that these two aspects of a system's construction may either be embodied in a single language or, as they prefer, in two separate, specialized languages. Such a coordination language is e.g., the Linda coordination model (see chapter 2.2.3).

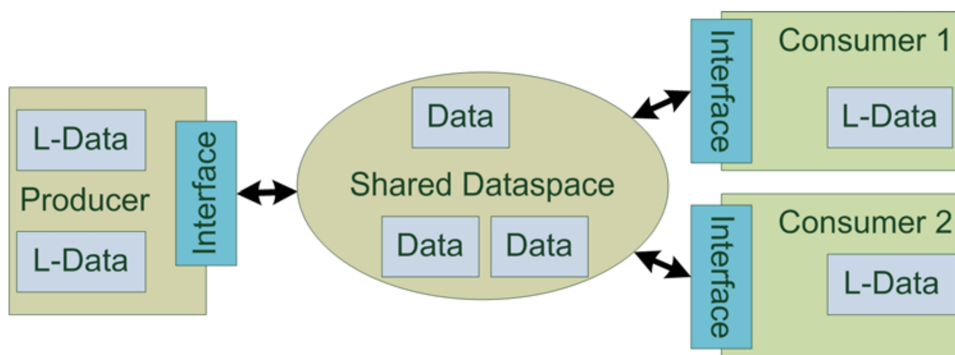

Figure 3: Data-driven Coordination

In the data-driven coordination model, processes exchange information by adding and retrieving data from a so called shared dataspace. Figure 3 shows three processes as in Figure 2. Local data stored within the processes, named L-Data, is not visible and exchanged data is placed into the shared dataspace and therefore visible to anyone that has access to it.

2.2.3. Coordination Frameworks

In the following relevant Linda related coordination frameworks are described. Each presented framework is described briefly including information about its coordination capabilities, its main intention, and distinctions to the Linda coordination model.

Tuple Space

The Linda coordination model [78] was developed in the mid-1980's by David Gelernter at Yale University. It describes the usage of a logically shared memory, called tuple space, together with a handful of operations (*out, in, rd, eval*) as a communication mechanism for parallel and distributed processes. In principal, the tuple space is a bag containing tuples with non-deterministic *rd* and *in* operation access. A tuple is built-up of ordered fields containing a value and its type, where unassigned fields are not permitted, e.g. a tuple with the three fields <“index”, 24, 75> contains “index“ of type string and 24 resp. 75 of type integer.

The defined operations allow placing tuples into the space (*out*) and querying tuples from the space (*rd* and *in*). The difference between *rd* and *in* is that *rd* only returns a copy of the tuple, whereas *in* also removes it from the tuple space. Both operations return a single tuple and will block until a matching tuple is available in the tuple space. There are also non-blocking versions of the *rd* and *in* operation, called *rdp* and *inp*, which return an indication of failure instead of blocking, when no matching tuple is found [231]. The *eval* operation is like the *out* operation, but the tuple space initiates a single or several threads and performs calculations on the tuple to be written. The result of these calculations is a tuple that is written into the space after completed evaluation and that can then be queried by other processes.

The Linda model requires the specification of a tuple as an argument for both query operations and thus supports associative queries, similar to query by example [241]. In such a case, the tuple is called template that allows the usage of a wildcard as a field’s value. A wildcard declares only the type of the sought field, but not its value, e.g. the operation rd(“index“ ?x, ?y) returns a tuple, matching the size, the type of the fields and the string “index“. A tuple containing wildcards is called an anti-tuple. If a tuple is found, which matches the anti-tuple, the wildcards are replaced by the value of the corresponding fields. The non-deterministic *rd* and *in* operation semantics comes from the fact that in case of several matching tuples a random one is chosen.

In the following Linda-like frameworks are described. They all introduce modifications in comparison to Linda:

ActiveSpace

ActiveSpace [2] uses objects rather than tuples for coordination. It provides a reduced API - consisting of a put and take operation - on top of a message oriented middleware

like Java Message Service (JMS) [150]. The features of such a JMS provider allow ActiveSpace to group spaces and to create subspaces.

In contrast to the Linda model ActiveSpace does not use templates to query the space, but an SQL-92 query string, whereby the result of the querying operation is placed into a subspace. Similar to JMS, ActiveSpace can operate in two different modes: queue and publish/subscribe. The former one allows only one consumer to retrieve the object that was placed in the space, while the latter one allows any consumer who is currently subscribed to the space to receive that object.

ATSpace

ATSpace [104], standing for Active Tuple Space, supports application-oriented matchmaking and brokering services. It provides a method of describing the way in which agents matching certain criteria are capable of finding each other in an open multiagent system. It gives agents the ability to perform application-oriented search algorithms on the tuple space. Therefore, the concept extends the Tuple Space API by the *find* method allowing each agent to supply its own matching algorithm without affecting other agents. ATSpace replaces the associative search for tuples by application-oriented search algorithms.

B-Linda

B-Linda [81] proposes an extension to eliminate the false matching phenomenon by adding an extended-type notion into the basic Linda model. The authors state that Linda was originally designed for closed environments where cooperation aspects concerned only one application. However, in open heterogeneous environments conflicts may occur, since the system is too big to be controlled. Therefore, the approach improves tuples to make them useable for heterogeneous applications. The concept targets to prohibit that a process not originally intended might read the tuple. It introduces the so called b-type of a tuple as a triple {St; Se; Sc} where St represents the structure of the tuple, Se its semantics, and Sc its scope in order to make more explicit the relationships between types and processes.

B-Linda structures tuples themselves to improve relationships between processes, but keeps the Linda coordination capabilities untouched.

Bauhaus Linda

Bauhaus Linda [163] generalizes the Linda model by eliminating Linda's distinction between a) tuples and tuple spaces by using the notion of mulitsets; b) tuples and templates by set-inclusions; and c) passive data and active processes allowing to handle both by means of the same primitives. Due to this generalization Bauhaus is simpler and more powerful than Linda, since coordination operations can be applied to the entire

space. However, distributed implementations of Bauhaus have to be capable of handling copies and migrations of processes.

Blossom

Blossom [222] is a C++ class library that implements a distributed tuple space supporting the use of multiple tuple spaces. Furthermore, it extends the Linda model by features like strongly typed tuple spaces to prohibit multiple tuples of the same type, field access patterns to make associative search more specific, or tuple space assertions for debugging purposes.

BONITA

In contrast to Linda, BONITA [192] provides asynchronous access, called BONITA primitives, to tuple spaces and thus allows user processes to perform computation concurrently. This leads to coordination constructs supporting more efficient programs. In the course of asynchronous operations BONITA introduces several more methods like *dispatch, arrived,* or *obtain*. Although these primitives improve the performance of coordination, since in some scenarios coordinated processes do not need to block, the coordination model itself has not been adapted. Further, it has to be made sure by the BONITA implementation that tuples appear in the tuple spaces in the same order as the originating process has produced them. Nevertheless, this is an implementation issue and does not affect the coordination model itself.

CORSO

Although Corso (Coordinated Shared Objects) [24, 69, 122] is based on the concepts of virtual shared memory [122] that allows the sharing of data objects between processes for coordination or information exchange, it is mentioned in this section because it offers Linda-like coordination capabilities. Objects are accessed via the object's methods to manipulate the internal state of the object; the new state of the object is published by *write* and retrieved by *read* methods. A blocking characteristic of these two operations is provided by using timestamps. If an object with higher timestamp than the timestamp of the operation is read then the operation blocks. The operation also block in case an eager replication mechanism of the objects has been selected. This means that operations are not blocked because of missing data in the object itself but due to the fact that the object is outdated, and the local copy of it has to be updated first by the replication strategy

DTuples

DTuples [108] introduces a Linda like peer-to-peer tuple space middleware build on top of Distributed Hash Tables. DTuples retains the basic features of the Linda tuple space while adapting them to the peer-to-peer system. The agents in the system see the system as repository of tuples. Management of storage of those tuples in the repository is the

responsibility of the system. The approach focuses on improving the efficient distribution and retrieval of tuples in a distributed environment rather than the coordination of processes.

eLinda

eLinda [229-232] improves the tuple matching mechanism. It enables the usage of more flexible queries, via its Programmable Matching Engine (PME), such as maximum or range queries. Beside these queries the PME also provides aggregated operations that allow the summary or aggregation of information from a number of tuples, returning the result as a single tuple. Although, the query expressiveness of the Linda coordination model can be improved with a PME, it only supports Linda primitives. Therefore, ordering information has to be embedded in the tuple itself and managed by the coordinating processes.

GigaSpaces

The GigaSpaces[5] implementation is a commercial product of JavaSpaces [74]. It is designed to be the core of a framework in which tuple spaces are used to guarantee high scalability and efficiency to applications. It also provides development frameworks not only for Java but also for .Net and C++. Beside the associative tuple matching mechanisms, GigSpaces offers a FIFO ordering of stored tuples.

GLinda

GLinda [117] combines the grid for job distribution and Linda for resource management. GLinda provides communication abstraction based on the tuple space concept on top of a grid-like environment to distribute workload of computationally intensive applications by using the Master/Worker paradigm. The point of this concept is to use the Linda coordination model in a distributed environment.

Grinda

Grinda (Grid+Linda) [36] is a tuple space implementation for the Globus Toolkit to be used by Grid applications as a coordination service. A major benefit of the Grinda implementation is the efficient indexing of tuples. In Grinda tuples are no longer ordered arrays like in the original Linda system but every class type is potentially a tuple. The approach suits better for XML messages used by the Globus Toolkit for the communication and requires less effort for the serialization.

---

[5] http://www.gigaspaces.com

Jada

Jada [176] is a coordination language for Java that extends the Linda coordination model by introducing new primitives for bulk operations, by replacing tuples with objects, and extending the matching policy. Beside the Linda primitives the new methods are *readAll, inAll, getAll*, and *getAny*. The first two return all those object which match the given template. The last two allow the user to define a set of templates which returns those entries that either matches all templates or any template. The matching policy in Jada can be extended by overwriting the provided *matches* method.

JavaSpaces

JavaSpaces [74] is the first specification of Linda in Java by Sun Microsystems. Like in Jada, tuples in JavaSpaces are objects implementing the *Entry* interface and stored in their serialized form. Consequently, two tuples match, if their serialized form matches. JavaSpaces extends the Linda model with (i) *rich typing* taking into account the field's type and the field's value (ii) *matching of subtypes* (iii) *lease time* indicating the tuples lifetime (iv) *transactions* (v) *notifications* informing registered processes about written tuples matching a specific template.

Kernel Linda

Kernel Linda [93] has been designed for supporting communication between operating systems' processes using the Linda coordination model. Each Kernel Linda process has its own associated memory space or environment. This environment uses QIX operating system to serve as an interface from the process to the rest of the world. In Kernel Linda, tuples are restricted to only one pair of key and value. However, Kernel Linda allows the implementation of multiple tuple spaces or dictionaries. Kernel Linda also has the ability to use a tuple space as a field of another tuple space. Kernel Linda can also work in a mixed language environment since it provides a set of language-independent data types.

KLAIM

KLAIM [20, 164-166, 183] is a programming language designed for programming mobile agents by supporting a programming style where processes and data can be moved across different computing environments. It supports programming distributed systems consisting of mobile components interacting via multiple distributed tuple spaces. The language uses explicit localities and allocation environments associating logical localities to physical nodes. It is made up of core Linda with multiple located tuple spaces and of a set of process operators. The spaces and processes are distributed over different localities. Operations are indexed with the location of the tuple space the operation is executed on allowing to distribute and retrieve data or processes over/from several different nodes directly. The so called Net coordinators describe the infrastructure, manage the distribution of data and processes, and set security policies

for accessing resources. KLAIM propagates a clear separation between the computational level (processes) and the Net coordinators allowing a more accurate handling of WAN applications.

The KLAIM communication model builds over and extends Linda's notion of generative communication [78] by introducing operation indices specifying the locations of the tuple space they operate on. However, KLAIM does not extend the semantics of the coordination itself and thus provides the same capabilities as Linda.

LIME

LIME [158, 182] assists system developers to deal with physical and logical mobility. Physical mobility involves the movement of mobile devices, while logical mobility is concerned with the movement of mobile agents, whereas both underlay the generic concept of a mobile component coordinating each other by means of transiently shared tuple spaces accessed via the basic set of Linda primitives. Movement results in implicit changes of the tuple space. The system is responsible for managing movement and the tuple space for restructuring associated with connectivity changes. One of the main concepts of LIME is reactive programming, the ability to react to events, like changes in the physical environment or changes in the quality of service. The run-time support continuously monitors the underlying layers of the virtual machine for system events and transforms them as event tuples into the LIME system becoming available to all other agents. LIME introduces the notion of a reactive statement, in which non-reactive statements are executed either synchronously or asynchronously in case a pattern matches a tuple. Additionally, reactions from all registered reactions are selected non-deterministically one after the other and the corresponding action is executed. Like in JavaSpaces reactions can be seen as logic executed due to a notification.

LighTS

LighTS [181] is a lightweight tuple space implemented in Java. Beside the basic Linda operations it provides building blocks (e.g. interfaces) to customize and extend its functionalities. Originally, LighTS was designed as the core underlying tuple space for the LIME coordination framework.

LuCe

LuCe (stands for Logic Tuple Centres [57-59]) introduces the concept of tuple centres as an extended tuple space, which can work as a *programmable coordination medium*. Beside normal tuples, information about the behavior of the centre is stored in the so called *specification tuples*. The main difference between a tuple space and a tuple centre is that the former supports only Linda coordination while the latter can be programmed to bridge between different representations of information shared by coordinated processes to provide new coordination mechanisms. Such mechanisms are realized by reactions allowing the extension of effects from the execution of communication

operations as needed. Reactions map a logical operation onto one or more system operations. Furthermore, the results of an operation can be made visible to the coordinating processes as a single transition.

LuCe extends the Linda coordination model by a dynamic coordination behavior realized by means of reactions. This allows LuCe to satisfy complex coordination requirements, like handling of ordered tuples. Reactions are limited to Linda primitives only. Therefore, they are only capable of handling coordination requirements which do not need the integration of other components for interaction than tuple spaces themselves. Beside the fact that reactions cannot perform blocking operations, to the best knowledge they introduce accidental complexity into the coordination framework due to missing structuring and separation of concern mechanisms. For instance, aggregation and ordering logic has to be implemented in one reaction. Furthermore, in case tuples need to be sorted according to a specific requirement, they have to be extended with additional information representing the current position of the tuple. This implies that every operation performed on the space has to be adapted to the new structure of tuples decreasing the overall performance of the system.

MARS

MARS [32-35] is a Java implementation of a coordination architecture focusing on mobile agents. Unlike Linda, the proposed architecture integrates a reactive model allowing manipulating the behavior of the tuple spaces by installing reactions. Reactions are called whenever mobile agents access the tuple spaces. This ensures flexibility in mobile agent applications, allowing the definition of specific and stateful coordination. A meta-level tuple space is introduced to manage reactions. Furthermore, the MARS interface adds two operations, *readAll* and *takeAll*, offering the capability to retrieve all the tuples that match with the template supplied as parameter. Reactions consist of 4 components: reaction (Rct), tuple item (T), operation type (O) and agent identity (I). The reaction is executed when an agent with identity I performs the operation O on a tuple matching T. A match in the meta-level triggers the corresponding reaction. In case of multiple matches, all corresponding reactions are triggered in the same order in which the matches occur. Reactions differ from the notify mechanism of JavaSpaces, which is simply a way to make external entities aware of matches in the tuple space. Notifications neither influence the built-in pattern-matching mechanism nor the semantics of the operations. The advantages of reactions are the homogeneous model for providing services in a tuple space, the implementation of specific local policies to adapt interactions to specific characteristics, and so simplifying the agent programming task compared with fixed-pattern matching interactions. MARS is similar to LuCe since it adopts programmable tuple spaces for mobile agent coordination. However, MARS exploits an object-oriented tuple space model that makes it more suitable to service and network management applications. As LuCe, MARS enables the

modification of the operations' semantics and thus extends the coordination capabilities of Linda but cannot influence the way how tuples are queried.

P4-Linda

P4-Linda [31] demonstrates the capabilities of Linda on top of the P4 portable parallel programming system designed to support portability across a wide range of multiprocessor/multicomputer architectures. Linda operations must adhere to a strict format, where a format string must be present as the first argument.

PLinda

PLinda [107] provides a set of extensions to Linda to support robust parallel computation on loosely coupled processors communicating over a network. The two principal extensions to Linda in PLinda are transaction mechanisms and process-private logging mechanisms which are integrated into transaction mechanisms enabling processes to restore a globally consistent state at rollback recovery, without any special consistency protocols. PLinda uses transactions for controlling concurrent access to shared data in a reliable manner despite failure and extends transaction functionality by supporting process resiliency without high runtime overhead. Robustness of Linda is achieved by replicating tuples on different storage media. PLinda supports two strategies: in the *Stable Tuple Space* all the updates made by the transaction are replicated on disk before a transaction commits, in the *Checkpoint-protected Tuple Space* the entire tuple space is periodically written to disk. As [107] describes, the advantage is that the difference between these two strategies is transparent to programmers.

PoliS

PoliS [42, 45] is a coordination model based on multiple tuple spaces, in which a tuple space consist of both tuples and other tuple spaces denoting a tree of tuple spaces. PoliS propagates two types of tuples: ordinary tuples and program tuples containing the coordination rules which manage activities inside the space they belong to. They can be read, consumed or produced just like ordinary tuples. Program tuples can influence a space tree by removing and adding tuples or spaces, but only those tuples of a space they belongs to or the tuples of the parent space. A reaction of a program tuple takes place if the space itself includes both the program tuple and a multi-set of tuples matching the pre-activation of the program tuple. Consequently, a local computation which does not modify the tuple space is performed. Afterwards the post-activation is executed. It is made up of a multi-set of tuples to be produced in its scope and of a set of spaces to be created. A constraint is that the tuples of the pre-activation must be consumed or read in the rule's scope. In case disjoint multi-sets of tuples satisfy triggering of a set of program tuples, those rules can be executed independently and simultaneously, since every rule modifies only that part of the space containing the

tuples that must be read or taken. Therefore other rules can alter other tuples in the space or in other spaces.

SwarmLinda

SwarmLinda [86, 145] is a prototype that demonstrates how abstractions, yielded from observations of swarms and the way they are organized, may be used to implement a scalable tuple distribution mechanism in a Linda system. The model used is based on ant colonies: the tuples are the food and the templates are the ants that try to find the requested tuples. During the search for tuples, the template releases a sort of trace on the visited nodes. This trace can be followed by next templates in order to optimize the search.

TSpaces

TSpaces [130, 131, 236] is a Java based tuple space system developed at IBM. It is based on the concepts of JavaSpaces but provides a different API to access the tuple space. TSpaces offers the possibility to query tuples by named fields or by specifying the field's index and a value or wildcard. Furthermore, TSpaces allows the definition of custom queries by introducing the concept of factories and handlers, and thus it is capable of supporting the storage of XML documents as tuple fields, which can be queried by using a subset of the XML Query Language (XQL). Besides the standard JavaSpaces operations, the API offers the operation *count* that simply counts the found matching tuples and returns an integer, or *delete* that removes a matching tuple from tuple space but in contrast to the *take* operation does not return it to the process. TSpaces also introduces a new operation called *rhonda* in its API. A process invoking the *rhonda* operation must pass a tuple as an argument. The tuple's field types are used as a template, which contains wildcards as values to find a tuple offered by another processes' *rhonda* operation. If two matching tuples are found they are swapped between both operations and the two involved processes will receive the other one's tuple. Furthermore, TSpaces allow the registration of processes to be notified in case tuples are written to or removed from the space.

TuCSoN

TuCSoN [52, 170, 171] has been inspired by the LuCe model and uses it to define an interaction space over nodes across the Internet, where each node represents a TuCSoN tuple center. The implementation is based on the LuCe architecture and addresses the issues of agent mobility. From the coordination capability's point of view, TuCSoN has the same advantages and limitations as LuCe.

XMLSpaces

XMLSpaces [219, 220] is an extension of the Linda tuple space model that adds support for storing and querying XML documents in a space. XMLSpaces are built on top of

TSpaces utilizing its extensibility to facilitate the storage of XML documents as tuple fields. As matching mechanisms XMLSpaces offers several XML query languages such as XQL or XPath.

XVSM

The eXtensible Shared Memory is the reference architecture of the Space-Based Computing architectural style and is described in detail in section 4.2.

2.2.3.1. Conclusion

Concluding, the listed Linda based coordination frameworks can be mainly categorized into a) frameworks using the power of the Linda coordination model in e.g., distributed environments or for coordination among operating system processes, or in combination with other technologies like Grid or Distributed Hash Tables; b) frameworks extending the matching capabilities of Linda; and c) frameworks extending the semantics of the coordination model itself.

Relevant implementations that support the exact tuple matching are: JavaSpaces, LIME, MARS, LuCe, and TuCSon. Although MARS, LuCe, and TuCSoN enable the modification of the operations' semantics by adding so called reactions, they do not influence the way in which tuples are queried. JavaSpaces add subtype matching to the exact tuple matching mechanism to query objects from the space. The drawback of exact tuple matching is that all collaborating processes must be aware of the tuple's signature they use for information exchange. Therefore, several tuple space implementations offer additional queries mechanisms, such as TSpaces, XMLSpaces, or eLinda. Both TSpaces and XMLSpaces support the use of XML-documents in tuple fields and therefore enable the use of several XML query languages. In addition, XMLSpaces use an XML-document like structuring for its space, which allows the utilization of sophisticated XML queries on the space. eLinda enables the usage of more flexible queries, via its Programmable Matching Engine (PME). Apart from GigaSpaces, it can be concluded, that all previously introduced tuple space implementations have in common that the stored tuples have no ordering. Furthermore, they do not guarantee which tuple is returned by a query, it may happen that due to the non-deterministic semantics of the Linda operations a tuple is never returned although it would match the query. Beside XVSM the only coordination framework that is capable of extending Linda's coordination capabilities is LuCe. A detailed discussion is provided in section 4.4.1. In the following, Table 1, summarizes the properties of the described coordination frameworks regarding its coordination and querying capabilities in comparison to Linda as well as the supported data format.

Framework	Coordination Capabilities	Querying Capabilities	Data Format
ActiveSpace	Restricted to inp and out operations	SQL-92	Java Object
ATSpace	Linda	customer-specific	tuple
B-Linda	Linda	eliminates false matching phenomenon	b-type tuple
Bauhaus Linda	Linda including passive data and active processes	set inclusions	multiset
Blossom	Linda	field access patterns, tuple space assertions	strongly typed tuple
BONITA	Linda with additional operation for asynchronous access	template matching	tuple
CORSO	write and read operations; blocking due to replication strategy	search for object identifiers	object
DTuples	Linda	template matching	tuple
eLinda	Linda	Programmable Matching Engine	tuple
GigaSpaces	Linda	template matching	object
GLinda	Linda	template matching	tuple
Grinde	Linda	template matching including class inheritance	object
JADA	Linda with bulk operations	overwriteable matching method	object
JavaSpace	Linda extended by lease time	Matching of subtypes	object
KernelLinda	Linda	template matching	limited tuple
KLAIM	Linda operations extended by location	template matching	tuple

	information		
LIME	Reactive Linda	template matching	tuple
LighTS	Linda	overwriteable matching method	tuple
LuCe	Customizable via tuple centre and specification tuples using reactions	template matching	tuple
MARS	Reactive Linda	template matching	tuple
P4-Linda	Linda	template matching	predefined tuple structure
PLinda	Linda	template matching	tuple
PoliS	Reactive Linda	template matching	tuple and program tuple
SwarmLinda	Linda	template matching	tuple
TSpaces	Linda extended by rhonda operation	customizable queries	tuple
TuCSoN	Customizable via tuple centre and specification tuples using reactions	template matching	tuple
XMLSpaces	Linda	XML querying	XML
XVSM	Customizable	Customizable	object

Table 1: Coordination frameworks and its coordination capabilities in comparison to original Linda

2.3. *Software Architecture*

This section gives an introduction to software architectures and describes the difficulties regarding software architectural styles concerning the implementation of changing requirements.

2.3.1. Software Architecture Definition

Whether or not explicitly known or even understood, every application has an architecture developed by at least one architect [217]. Software architecture deals with design and realization of software systems. The architecture of a system is not a phase of development but rather the result of a particular phase in the development process where key design decisions on requirements were made. Design decisions have an effect on the entire development process and may refer to conclusions regarding e.g., the system's structure (the organization of its elements), its functional behavior, interaction requirements, non-functional properties, or its implementation. Architecture is defined as the set of design decisions on a system and thus it reflects the spirit of an application. Concluding, a widely accepted definition of software architecture [234] is that "*The architecture of a software-intensive system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them*". Another definition[6] taken from the IEEE community refers to "*Architecture is the fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution*".

Although the first definition is general, it already contains important aspects of software architecture: system structure, elements, the characteristics of an element, and the relation between the elements. Perry and Wolf pick it up and simply define architecture as the set of elements, form, and rationale [179].

According to [179], elements are the main artifacts of the system to be realized, like essential classes, interfaces, components, frameworks, subsystems, or modules and are grouped into so called building blocks. Such blocks are captured according to the services they provide: processing elements, data elements, and connecting elements. These three types are usually consolidated into two major architectural concepts: components (computational elements) and connectors (interaction elements) for which the following definition can be derived [217]:

- Software Components: Software components are architectural elements, which a) encapsulate data and processing, b) restrict access via explicitly defined interfaces, and c) have explicitly defined dependencies on their execution context. A component can be simple or complex as an entire system but only visible via the interface made public.
- Software Connectors: Software connectors abstract the task of managing interaction between the various software components. A connector might be a procedure call, or a shared data access. While components provide application

[6] ANSI/IEEE Standard 1471-2000, Recommended Architectural Description of Software-intensive Systems

specific services, connectors do not. Characteristics of a connector can be discussed without an application specific context. For instance the publish/subscribe paradigm can be built without a specific scenario, and then be used in several applications.

The form describes the way in which the elements are organized and interconnected in the architecture. It represents the structure of individual elements, how elements are composed in the system, how elements are interacting with each other, and the relation of elements to the operating environment. The composition of elements is also known as configuration or topology. The rationale represents the system designer's intents, assumptions, choices, external constraints, non-technical constraints, selected patterns and styles, and any other information that cannot be derived easily from the architecture. The rationale defines why the architecture was built the way as it is looks like now.

As already mentioned, a main aspect of software architecture is that it is created by a set of principal design decisions. As the architecture of a software system is developed and evolves, new design decisions are made, old ones removed, or changed. Therefore, the architecture of a system is a temporal aspect [223].

2.3.2. Software Architecture Concepts and Principles

Software architectures have the characteristic of either being prescriptive or descriptive [217]. On the one hand, a prescriptive architecture is the combined set of design decisions made reflecting the intent of the architecture. Therefore, a prescriptive architecture is the system's as-intended architecture. On the other hand, a descriptive architecture is the set of realized design decisions, and thus represents the system's as-realized architecture. With every decision made new prescriptive or descriptive software architecture is created, where each pair represents the system's architecture at a given time. In an optimal scenario the two sets would be identical. A difference may occur for instance when using an off-the-shelf middleware platform implementing various design decisions that influence design decisions made for the system under development. There is a trend to use COTS[7] software and thus to introduce large amounts of functionality into systems which is not needed, but which may interfere with the intended functions [144]. The difference between prescriptive or descriptive architecture is also called architectural degradation introducing two more terms, namely architectural drift and architectural erosion. Architectural drift, the result of direct changes in architectural elements, is a new design decision in the descriptive architecture that neither exists in nor violates any design decisions in the prescriptive architecture. A drift can lead to a

[7] Commercial off the Shelf

loss of clarity and understanding, and if not properly addressed, it will eventually result in architectural erosion. Architectural erosion is a new design decision introduced in the descriptive architecture that violates design decisions in the prescriptive architecture.

As explained software architecture is mainly about elements of the software system and their interactions. These components realize the functional requirements of a software system. However, of the same importance are non-functional requirements like performance, costs, maintenance, reusability, or reliability. These factors have a great influence on the structure of the architecture of a software system. There are several principles [62, 203, 223, 234] that help to create a "good" architecture. The main focus is on the reduction of architectural complexity and increasing its flexibility:

- Loose coupling: Coupling between architectural elements should be minimized as much as possible. This principle deals with the issue that to understand or change an element another one needs to be understood or adapted as well. Therefore, the purpose of loose coupling is to minimize the structural complexity by minimizing mutual dependencies. The lower the coupling between elements the easier it is to understand and change one element without the need to understand or change other one.
- High cohesion: Coupling deals with dependencies between elements of the architecture, cohesion with dependencies between subparts (e.g., variables, data structures, methods) of such elements. High cohesion is desirable since an element unites all of its subparts and thus it is easier to understand and to change the element itself. Architectures making use of the high cohesion principle allow elements to be interpreted as black boxes which can be manipulated and adapted independently of other elements.
- Design for Change: This principle refers to the fact that software systems are always changing and new requirements are hard to foresee. The wish is that the architecture is designed in a way that supports predictable changes, like functionality which has not been implemented yet due to high costs or time pressure. Such requirements can be assessed in advance and taken into consideration by looking out for similarities in previous projects.
- Separation of concerns: This is the process of separating different aspects of the same problem in order to be capable of handling each of these aspects independently. This allows a complex system to be divided and structured into smaller independent component parts which are better understandable and manageable than the entire system. This means, that the parts of the software system responsible for specific aspects of the problem descriptions have to be identified and realized in loosely coupled individual modules.
- Information hiding: The principle of information hiding helps structuring and understanding complex systems by providing only those parts of information

which are really needed by other elements whereas remaining parts remain hidden.

- Abstraction: Abstraction is a special form of separation of concerns by separating the important from the insignificant aspects of a complex problem.
- Modularity: Software architecture should consist of well-defined elements with definitive responsibilities. This supports adaptability and reusability. This principle is a combination of the aforementioned abstraction, separation of concerns, and information hiding. Approaches which support modularity in software systems are e.g., object-oriented programming, component-based architectures, or layered architectures.

2.3.3. Software Architectural Styles

With respect to the given context certain design decisions result in architectural solutions with good properties. These kinds of decisions can be seen as lessons learned from experienced software engineers forming so called architectural styles. Architectural styles do not entirely specify components, interactions between components, or their configuration but provide means for justification to state the rationale that underlies them. According to [217], "*an architectural style is a named collection of architectural design decisions that (1) are applicable in a given development context, (2) constrain architectural design decisions that are specific to a particular system within that context, and (3) elicit beneficial qualities in each resulting system*".

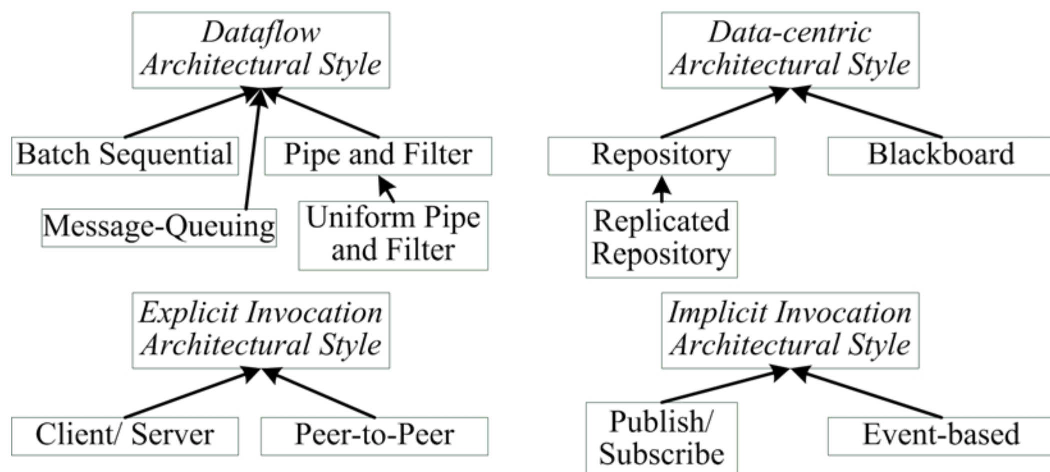

Figure 4: Architectural styles and their categorization

New architectures can be defined as instances of specific styles [63]. Since architectural styles focus on solving very specific types of problems, a given architecture may

combine a number of styles to meet design problems [234]. However, a good designer should select a style that matches the needs of the particular problem being solved [200]. In the following four architectural styles [11, 193, 201] (Figure 4) with specific focus on interaction and adaptability capabilities are described.

2.3.3.1. Dataflow Architectural Style

Dataflow architectural styles are concerned with the movement of data between independent processing elements. These styles are in particular useful in case complex tasks can be divided into several simple subtasks which can be defined as a series of independent computations.

Batch Sequential

Batch sequential is the oldest dataflow architectural style. Programs are divided into separate sub-programs which are executed in order. Data is passed from one subprogram to the next where it is aggregated and passed on to the next subprogram. In the batch sequential architectural style the processing steps are discrete in the sense that each step finishes before the next step may commence.

Pipe and Filter

In the pipe and filter architectural style a stream of character data is passed incrementally between each component. A component is a filter transforming streams of incoming data into streams of outgoing data. Each filter reads the stream of data on its input and produces a stream of data on its output while applying a transformation step to the data. The connector is the pipe between the filters (Figure 5). In contrast to batch sequential filters have no limitations on a producing component to finish before a component that consumes the producer's output begins.

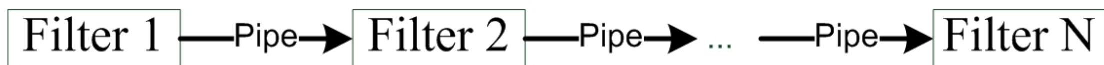

Figure 5: Pipes and filter architectural style

The advantage of the style is that filters are independent components which can be combined easily by means of pipes and thus the designer understands the overall system as a simple composition of the behaviors of the individual filters. Since components are independent, pipes and filters can be executed concurrently increasing the overall system performance. This also improves reusability since any two filters can be connected. However, these filters need to agree on the data structure being transmitted. New filters can be added; old ones removed, or updated easily supporting maintenance and evolvability of the system.

Attention has to be paid in case complex data structures are transmitted between the filters since it introduces additional overhead due to transformation into common data formats, in case a filter cannot incrementally process incoming data, or in case interactions between filters are required to share state information.

Uniform Pipe and Filter

The uniform pipe and filter style extends the pipe and filter style by introducing the constraint that every filter has to have the same interface. The benefit of this style is that it allows the combination of independently developed filters and minimizes the knowledge needed to understand a filter.

Message-Queuing

Queuing (Figure 6) is necessary for instance, when temporal outages of the receiver should be tolerated. Messages are not passed from one component to the other directly, but through intermediate message queues that store and forward the messages. Consequently, the sender component and the receiver component are decoupled without the need to know each other's location or identity or be up and running at the same time. This means that the sender component puts messages into a particular queue and does not necessarily know who consumes the messages and when it is consumed.

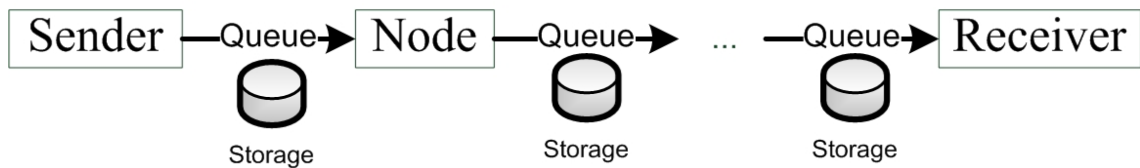

Figure 6: Message-queuing as a dataflow architectural style

2.3.3.2. Data-centric Architectural Style

The essence of data-centered styles is that multiple components have access to the same central data store, and communicate through that data store.

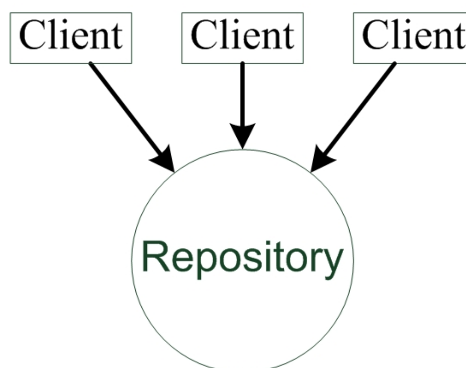

Figure 7: Example of a repository based architectural style [11]

Repository

A shared repository provides its clients access to shared data by means of an API or query language. The shared repository has to make sure that access to data is performed efficiently, that it is scalable, and that it can assure the consistency of the stored data. The difference to dataflow architectural styles is that design attention is explicitly paid to the structure of the repository. However, a repository does not provide the means for specifying in which order its shared data needs to be processed by its clients. Databases are the typical representation of this data-centered architectural style.

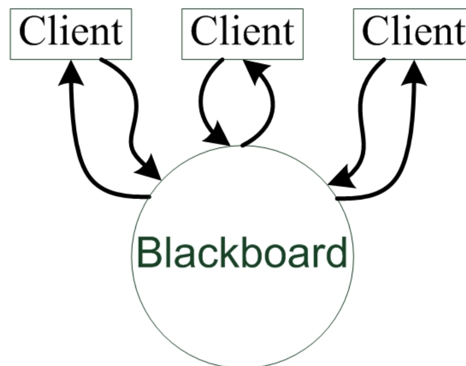

Figure 8: Example of a blackboard based architectural style

Blackboard

Active repositories tie together the shared repository with another architectural style, termed event-based systems [38]. An active repository, called blackboard, is able to notify registered clients about changes. The state of the information on the blackboard determines further processing.

In [72] an extension to the pipes-and-filters style was proposed, where a shared repository is supported. However, the hybrid framework does not offer the abstraction of the pipes-and-filters style but rather adds shared data to the pipeline. The advantage of the hybrid solution is that it enabled an explicit and consistent modeling of feed-back loops

Replicated Repository

The replicated repository architectural style improves the accessibility of data and scalability of services of a system by running several processes which provide the same service. These processes interact in order to present the illusion of a centralized service.

The main advantage is performance gained by reduced latency and due to execution of disconnected operations in case the main server has crashed. Maintaining consistency between the running processes is the primary concern.

2.3.3.3. Explicit Invocation Architectural Style

The explicit invocation architectural style [11, 217] is characterized by calls that are invoked directly between the components of the architecture.

Client/Server

This style consists of two components: client and server. Clients are independent and send requests to the server by means of e.g., remote procedure calls, representing the connector between the components. The responsibility of the server is to provide the service the client asked for. The style accommodates the addition and deletion of clients but lacks performance and scalability if there are a large number of client requests. Furthermore, adaptations on client side have to be performed in case there is a switch from a single server to multiple servers.

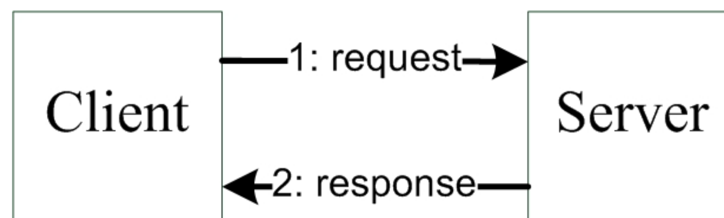

Figure 9: Client/Server architectural style

Peer-to-Peer

An alternative to the client/server architectural style is the peer-to-peer style [208] where both information and control is distributed. Peers directly communicate with each other and combine the roles of a client and a server. Ideally, this structure can be operated without the need for a centralized server component. However, most of the current systems rely on a combined approach, where particular services are hosted by servers explicitly. A limitation to this approach is given by the network bandwidth and the latency needed for information retrieval.

2.3.3.4. Implicit Invocation Architectural Style

The implicit invocation architectural style is characterized by calls that are invoked indirectly and implicitly as a response to a notification or an event. The benefit is improved adaption capabilities and enhanced scalability due to indirect interaction between the loosely coupled components. The group is represented by the publish/subscribe [68] and event-based [136] architectural styles.

Publish/Subscribe

The publish/subscribe architectural style defines two types of clients: publishers generating information, and subscribers receiving notifications of data they have previously subscribed for. This type of messaging paradigm allows a strong decoupling of publishers and subscribers in time and space. Furthermore, it enables asynchronous and anonymous communication (decoupling in reference) between publishers and subscribers [68]. Exemplary system implementations realizing this architectural style are illustrated in section 5.1.3.3.

Event-based

The event-based style refers to independent components entirely communicating by sending events via the event-bus. In its simplest form every generated event is transmitted to all other components which can decide whether to react to the event or to ignore it. In more sophisticated implementations, events are only transmitted to components which are really interested in them. This makes event-based systems similar to publish/subscribe systems. The difference is that in event-based system every component is equal and no distinction between publishers and subscribers is performed.

2.3.3.5. Complex Software Architectural Styles

According to [217] complex architectural styles are those which provide greater benefit and are specialized to a certain application context. In the following the C2 architectural style and the distributed objects style are described exemplary.

C2 Architectural Style

Although the C2 (*C*omponents and *C*onnectors) architectural style's aim was to bring the model-view controller pattern [62] into the context of distributed and heterogeneous environments, it became beneficial for a lot of applications and not just GUI application domains. A system may need to support requirements [11] such as different implementation languages of components, diverse GUI frameworks, distribution in heterogeneous networks, concurrent interaction of components without shared address spaces, or multi-user interaction, without violating performance requirements and supporting a clean separation of concerns. The C2 style joins concepts from layered [30] and event-based architectures.

The structure of the C2 style refers to a top-to-bottom hierarchy of concurrent components that interact asynchronously by means of sending messages through explicit connectors. Components submit requests upwards in the hierarchy, knowing the components above. Downwards in the hierarchy notification messages are sent without knowing the components beneath. Each component has a top and a bottom part. The top

part specifies the set of notifications it wants to react to, and requests it will send. The bottom part specifies the set of notifications it may send and the requests to which it responds. Components are only connected with connectors, but connectors may be connected to both components and other connectors. The purpose of connectors is to broadcast, route, and filter messages.

A benefit [217] of the C2 architectural style is the easy adaptation of the application to cope with new platforms (operating systems, interface toolkits) by enforcing independence. Furthermore, the heterogeneity of components enables applications written in different programming languages and running on different platforms distributed in the network. It supports concurrent components both being capable of running on a shared processor or on multiple machines. Additionally, it supports network distributed applications since connectors abstract details of communication protocols. Although each of the benefits is also represented in simple architectural styles, the contribution of the C2 architectural style is the combination of selected simple architectural styles into a coherent comprehensive approach [217].

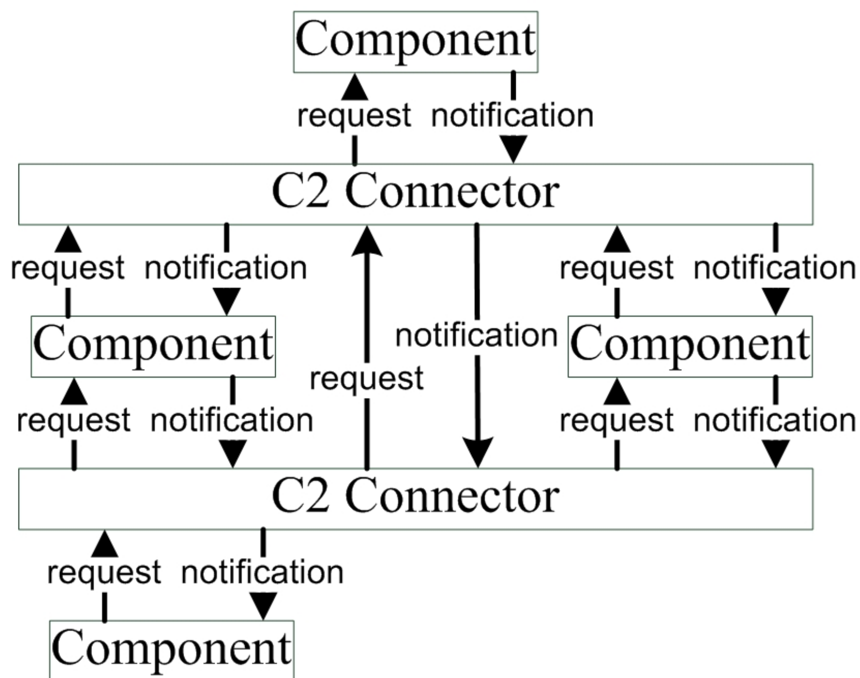

Figure 10: An example of a C2 architecture with four components in three layers, and two connectors that delimit the layers

However, the C2 architectural style poses some limitations to be considered. Applications are layered networks of components interconnected by messaging connectors with the primary responsibility of routing and broadcasting them. Therefore, interaction capabilities have to be implemented with connectors. The introduction of layered filtering of messages solves problems with scalability, but routing across multiple layers can be inefficient.

Distributed Objects

Originating from the object-oriented style distributed objects want to bring the benefits of the object-oriented style into a distributed and heterogeneous environment. In this style objects are instantiated at different nodes in the network each exposing a public interface. The components in this style are objects representing application functionality while connectors refer to remote procedure calls. State is distributed among the objects which can be advantageous in terms of keeping the state where it is most likely to be up-to-date. However, it has the disadvantage that it is difficult to obtain an overall view of system activity [193]. The object's interface must be serializable and supports synchronous procedure calls, like in CORBA [94]. Furthermore, there must be some controller object that is responsible for maintaining the system state e.g., in case objects fail or change their identity [77]. The style combines the client-server style (to provide the notion of distributed objects) and the pipe and filter style (to overcome the problem of cross-machine communication) [217].

2.4. *Software Evolution*

Software architectures are a challenge in terms of software development and evolution. Yet it is known from experience that evolution is a key problem in software engineering and demands huge costs. Companies spend more resources on maintenance (i.e. evolving their software) than on initial development [106]. Software evolution refers to the implementation of changes in a software system. Experiences [204, 205] gained by several case studies revealed the most influential risks and their affects on the design of the software architecture. The range of affects starts at adding architectural add-ons (due to e.g., insufficient requirements negotiation) and ends at a complete refactoring of the architecture (due to e.g., poor clustering of functionality). This section describes needs for adaptations regarding software architectures, details types of changes, illustrates architectural styles supporting changes, and explains advantages and limitations of agile software development.

2.4.1. Motivation for Change

There are several different motivations for changing software architectures [217], as listed and explained below:

- Corrective change: the application is adapted in order to equal descriptive architecture with prescriptive architecture. This means that after adaptation the application conforms to existing architectural requirements.

- Modification to the functional requirements: new features are introduced, and existing ones modified or removed. In most of the cases such changes become relevant due to the success of the application.
- The introduction or change of non-functional system properties like security, performance, replication, or scalability may induce significant changes to the system's architecture.

2.4.2. Adapting Architectural Elements

Any of the system's architectural elements may be subject of adaptations. The difficulty of realizing change requests depends on the characteristics of the elements themselves.

Component

Changes may affect internal parts of a component while the interface of that component remains untouched. Such changes may need to be performed in order to e.g., improve the performance of the component by replacing an internal algorithm or simply correct implementation mistakes (i.e. bugs).

Another type of changes refers to the specification of the component that has the knowledge of self. The component may collect information about itself and make it available to the outside world. For instance, run time assertions validate incoming parameters and collect them if they are violated. Based on this information the software architect may identify ways in which the software architecture in general needs to be modified. In case the component's knowledge about its specification is not only available to the component itself but also to other components, then the system is capable of dynamically selecting components meeting system goals and thus this approach strongly supports adaptations. In case a component has knowledge about its role in the software architecture it may monitor other components providing the same service. Based on the collected information the component may change its behavior in case the monitored component has crashed by requesting to be replaced.

Component Interface

Changing a component's interface usually cannot be avoided if the component's functionality has to be adapted. This in turn implies changes to all components which use the interface.

An approach to cope with this issue is to introduce so called adaptors which emulate the previous interface and forward incoming request to the new one. However, due to the increased number of components and interactions this implies that the complexity of the architecture is increased and tracing interactions to understand the system's functionality becomes more difficult.

Connector

Changes of connectors [172, 173] may be fundamental to the architecture of a software system. In most cases connectors are changed due to altered non-functional properties like distribution or performance. Connectors mediate and govern interactions among components, and thereby separate computation from communication, and thus minimize component interdependencies. Components must not communicate by directly referencing one another. Rather they shall utilize connectors, which localize and encapsulate component interfacing decisions. This minimizes coupling between components, enabling to change binding decisions without requiring component modification. Connectors that support loose coupling between the components (e.g., event-based architectural styles) are good candidates for supporting architectural changes. Additionally, connectors can implement different change policies by altering the conditions under which newly added components are invoked. For example, to support immediate component replacement after a certain point in time, a connector can direct all communication away from the old component to the new one.

2.4.3. Architectural Tactics supporting Adaptations

In the following tactics [234] to ease management of changes in software architectures are summarized.

Contain Change

Changes applied to architectural elements become only a problem when they trigger changes in other architectural elements. An example is the change of a component interface as described in the previous section. The challenge is to design a system where changes are restricted to small, well-defined parts of the system. Design principles like encapsulation, separation of concerns, or single point of definition are design principles which help localizing the effects of a change.

Variation Points

In order to support certain kinds of changes, the identification of specific locations in the system's design, called variation points, helps to adopt localized design solutions. Mechanisms to realize the requested change are for example:

- A variation point is a location where elements can be replaced with alternative implementations. This requires the usage of interfaces separating it from the specific implementation. The system's behavior is changed by replacing the implementation.

- Certain aspects of processing (e.g., inputs, outputs) should be parameterized by means of configuration parameters to allow the change of system's operation without modifying its implementation.
- The separation of physical and logical processing may represent another variation point. For instance, data formats changing frequently should be processed first in a component that copes with the physical format and then forwarding the result to the business logic.
- Another variation point is to break a monolithic piece of software into several processing elements. The variation point represents the structure of processing elements that can be easily manipulated.

Change-oriented Architectural Styles

Beside architectural styles particularly effective in supporting changes, techniques facilitating change can be categorized [217] into interface-focused ones and architecture-based approaches.

Interface-focused architectural solutions do not support all types of changes; in fact they focus on adding functionality in the form of modules:

- Application Programming Interfaces (API): as already described in the previous section, new modules can be integrated into the system by implementing a specific interface and thus replacing the old module. This type of change is limited to the functionality the API already offers.
- In plug-in based architectures ("*mirror-image*" of API [217]), the original application defines interfaces to be used by third-party components. Instead of calling application specific interface methods of the added component, the original application calls out to the added component. However, the added application must be aware of plug-ins to support this technique.
- In component/object architectures, the original application publishes its internal interfaces for adaptation purposes to third party developers. In contrast to the API and plug-in based approaches (where the application is seen monolithic), here internal components are exposed and subject to changes by replacing the implementation behind the replaced component interface. This approach is more powerful in comparison to the two previous ones since more interfaces are exposed. However, developers have to deal with source code, rather than architectural models.
- Scripting languages can be used to implement add-ons executed by an interpreter to alter the behavior of the application. Such languages usually provide domain-specific language constructs supporting the implementation of add-ons.

In case interface-focused solutions are not powerful enough to satisfy changing requirements (i.e. more than just adding new modules), architecture-based approaches are needed. A discussion in [217] about architectural styles reveals that the main aspect in architectures regarding change implementation relates to bindings. Architectural styles enabling easy addition and deletion of components facilitate adaptation. This can be achieved by using architectural styles consisting of connectors which enable independence between components, like the event-based connector. Complex dependencies between architectural elements and inflexible connectors may hinder solving any adaptation problem.

Chapter 3

3. Research Contribution

This chapter describes the research contribution by explaining the thesis' research issues, identifying the used research methods and concepts for evaluation. Additionally, it describes the scenarios used for evaluation, namely the Traffic and Transportation scenario (RealSafe), the Air Traffic Management scenario (SWIS), and the Production Automation scenario (SAW).

The target of this thesis is software developers developing complex distributed applications. Figure 11 shows the problem statement with a given distributed software system consisting of several application components. These components need to coordinate with each other to achieve the system's intention. Furthermore, the architecture of the software system is subject of changes due to new business or infrastructure requirements.

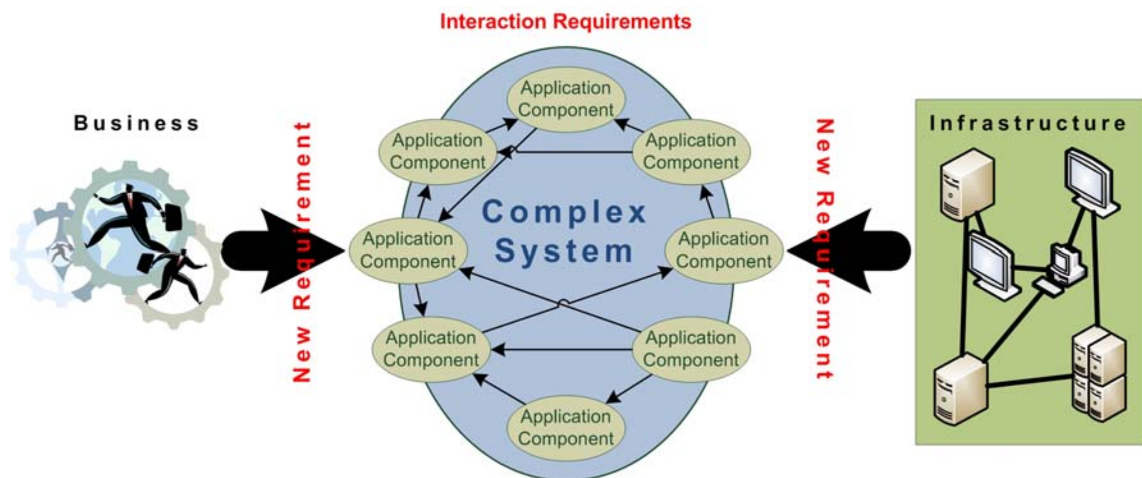

Figure 11: Overview of the research challenges

The major challenges for software developers are a) meeting coordination requirements of the application, b) increasing the software architecture's flexibility to be prepared for future requirements, and c) minimizing complexity of software architecture and application components.

For these challenges, the novel Space-Based Computing paradigm is suggested to support software developers managing complexity issues regarding interaction requirements and agility of the software architecture. This paradigm defines a new architectural style (that we term the Space-Based Computing architectural style or SBC for short) that flexibly combines and abstracts the properties of several architectural styles and extends them by sophisticated coordination models. It is the overall

contribution of the thesis to show that in contrast to traditional coordination frameworks, SBC supports exchangeable coordination models and regardless of the evolutionary degree or state of the system, it offers software developers a stable, manageable, and easy understandable set of architectural concepts to work with.

3.1. *Research Issues*

This section identifies the research contributions addressed in this thesis which all refer to the SBC architectural style and its contribution towards interaction and agility.

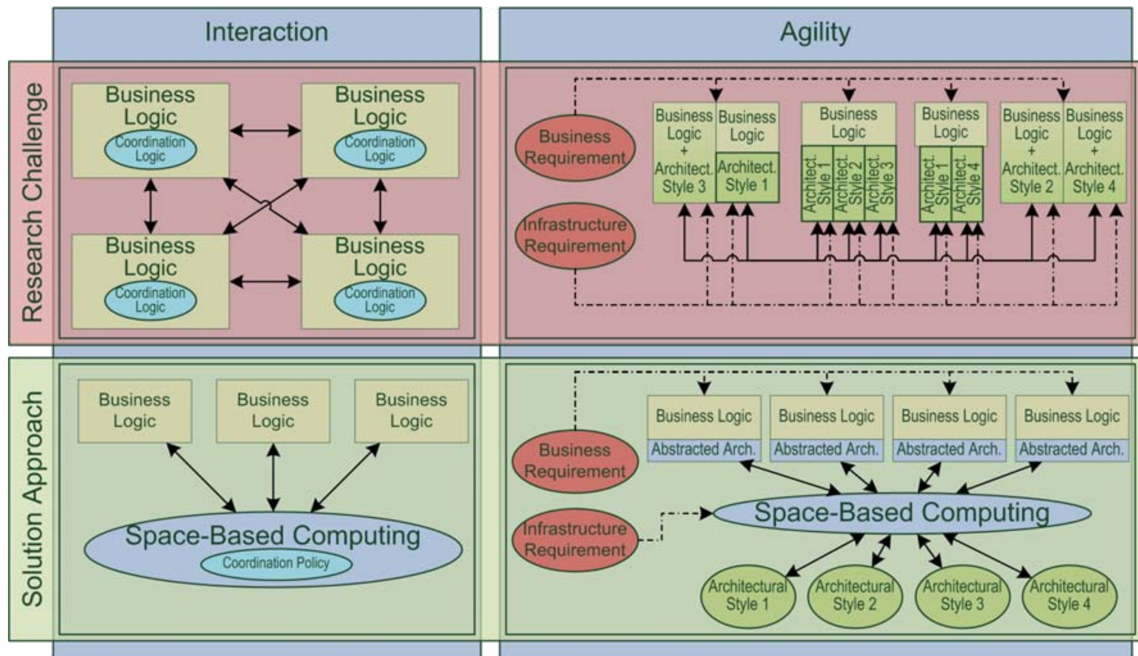

Figure 12: Overview of research issues.

Figure 12 shows the two contributions concerning interaction and agility: The upper part of the figure presents the research challenges and the lower part the solution approach. The left side of the figure depicts the first research contribution dealing with flexible interaction mechanisms and the right side with software agility meeting changing requirements.

The left upper part shows the high number of interconnections necessary in control-driven coordination models which increase with the number of participating application components. The solution approach (lower part) suggests an abstraction of coordination details through flexible and interchangeable coordination policies. This way, application components only need to select the right coordination policies and can therefore be liberated from knowing the details of coordination implementation. Additionally, the complex reintegration of failed and recovered components into a still running coordination process is simplified.

The right side of the figure deals with the question how changing requirements can be efficiently handled. The upper part shows application components consisting of business logic using a traditional architectural style. In some components business logic and architectural style are clearly separated by means of well-defined interfaces whereas in others they have become intertwined. In both cases - even when clear interfaces are used - there is a certain dependency (section 1.2 and 2.3) between business logic and the architectural style. The proposed solution approach suggests SBC as an architectural style abstracting various architectural styles thus reducing the complexity in application components.

3.1.1. Interactions in Complex Software Systems

For the interaction of application components either the control-driven or the data-driven coordination model can be used (see section 2.2.2). Message-oriented middleware, a representative of the control-driven coordination model, is a widely used middleware platform for the implementation of distributed applications, since it allows a strong communication decoupling (concerning time, space, and reference [68, 101]) between application components. However, coordination requirements beyond point-to-point, publish/subscribe, or request/reply communication modes which cannot be handled by the messaging interface only have to be explicitly dealt within application components. The software developer has to implement additional coordination logic which increases the complexity of these application components. The Linda data-driven coordination model is well suitable for coordination requirements but lacks performance in case of coordination issues with ordering requirements (e.g., FIFO order) and introduces additional dependencies due to its static coordination model. Based on this, the following research contributions (C) are derived:

C1.1: **Space-Based Computing bridges control- and data-driven coordination.** SBC offers the software developer a coordination abstraction via a simple API that subsumes both coordination models. SBC hosts flexible and exchangeable coordination policies which represent control- and data-driven coordination models and integrates their advantages and compensates their limitations. The thesis discusses the differences of the two models in relation to SBC and based on use cases investigates the effectiveness of this claim in sections 4.3, 4.4 and 6.2.

C1.2: **Space-Based Computing improves coordination efficiency.** SBC addresses various kinds of coordination requirements and supports the clear separation between business logic and coordination logic. In contrast to tuples in Linda coordination model SBC further distinguishes between plain payload and coordination data. Coordination policies in

SBC make use of this distinction to coordinate application components efficiently, thus contributing to an overall efficiency improvement. The issue is addressed in sections 4.2.1.2 and 4.2.4.2 while in sections 6.1.1 and 6.1.3 use case scenarios are used to set up benchmarks evaluating SBC in comparison with the Linda coordination model and queues.

C1.3: **Space-Based Computing facilitates continuous coordination.** SBC facilitates zero-delay reintegration of failed and recovered application components into the still running coordination process, thus enabling a robust continuous coordination capability. SBC stores coordination data in a structured shared data space allowing coordinating application components to maintain a global view of the state of the coordination process. The claim is addressed in sections 4.2.1.4, 4.2.2.1, and 4.2.4.4, and evaluated by means of benchmarks and process models in sections 6.1.1.3 and 6.1.3.1.

C1.4: **Space-Based Computing reduces coordination complexity in applications.** The clear separation between business logic and coordination logic, and the representation of flexible and powerful coordination policies requires from applications to know only the coordination policy and its interface, rather than an exact implementation of the coordination policy, thus minimizing the complexity of applications. Based on a literature review regarding coordination frameworks and control and management of complexity, and based on evaluations of application scenarios using process models the claim is investigated in sections 4.2.4, 4.3, 6.1.1.4, 6.1.3.3, and 6.2.

3.1.2. Evolution of Complex Software Systems

Software systems are subject to changes. Such changes are triggered because of changing business or infrastructure requirements, which the software system has to cope with. Agile software development especially targets the issue of changing requirements from a software development perspective, and pronounces to respond to changes rather than following a software development plan. However, it is essential that the underlying software architecture is capable of managing changing requirements, since criticism on agile software development states that it lacks paying attention to architectural and design issues and therefore is bound to engender suboptimal design-decisions. Software developers have to be careful about selecting the proper architectural style and thus the usage of an appropriate middleware technology representing the chosen architectural style for a given problem statement; otherwise non-functional properties may suffer. In case new requirements demand a switch to or an integration of other architectural styles, additional cognitive complexity is introduced to the application. Consequently, instead

of a stable set of architectural concepts for effectively managing complexity concerns, the number of concepts a software developer has to work with explicitly increases with the size and degree of evolution of the system. Software developers implementing application components seek for frameworks abstracting these complexity issues allowing them to entirely focus on business goals. Based on this, the thesis derives the following research contributions:

C2.1: **Space-based Computing supports flexible software architectures.** SBC is extensible and flexible by supporting the integration of orthogonal concerns (i.e. non-functional features) without changing the application. SBC decouples all non business related issues from business specific goals by clearly separating concerns (e.g., replication, migration, and location) related to a distributed system. This allows software developers to focus on specific concerns in case software changes and to treat them isolated. The claim is discussed and investigated by means of the so called XVSM reference architecture in the sections 4.2, 6.1.1.2, and 6.1.2.

C2.2: **Space-based Computing facilitates robustness against changing requirements.** SBC abstracts and captures the characteristics and properties of various architectural styles in a single API. This allows the migration to or the integration of other architectural styles transparently to the application avoiding architecture breakers. The claim is addressed in sections 4.2.4 and 4.3. Application scenarios present requirements which are evaluated in sections 6.1.1.2, 6.1.2, and 6.1.3.1.

C2.3: **Space-based Computing reduces complexity in applications introduced due to changing requirements.** SBC is a flexible concept that captures the properties of various architectural styles and decouples responsibilities regarding the handling of distributed system related issues. Based on a literature review regarding the development of flexible and dynamic architectures the claim of reducing complexity in application components due to changing requirements is investigated in sections 4.2.4, 5.2, 6.1.1.2, 6.1.2, and 6.1.3.1.

3.2. *Research Methods and Evaluation Concepts*

This section describes the research methods used in this thesis as well as the evaluation concepts and evaluation criteria.

3.2.1. Research Methods

In order to define and address problem statements, and position the research contributions, we carried out interviews with well-known researchers and a systematic literature review [27] on complexity in general and complex systems (see section 2.1), coordination theory and coordination frameworks (see section 2.2), software architectures (see section 2.3), and software evolution (see section 2.4).

For feasibility evaluation of the research claims, we used several different methods. We realized prototype implementations as proof-of-concept [71] of our conceptual approaches. The goal of the prototyping process is the identification of processes which involve an early practical demonstration of relevant parts of the desired software. Many software developers are motivated to employ prototyping by important conclusions drawn from their working experience. Furthermore, we conducted usability studies by means of students implementing exemplary scenarios in lab courses at the Vienna University of Technology. Additionally, we performed theoretical proofs by means of architectural comparison and analyzes. Process models helped for comparing the number of execution steps needed to achieve a certain goal. For performance evaluation, we follow the guidelines for empirical research in software engineering [16]. The guidelines are intended to assist researchers, reviewers, and meta-analysts in designing, conducting, and evaluating empirical studies.

3.2.2. Evaluation Concept

For investigating the collected research claims a set of general requirements based on the description of the used application scenarios from different industrial domains were gathered. The next step involved the adaptation and implementation of the SBC architectural style (realized in the XVSM reference architecture) based on special requirements of particular use cases (chapter 5). For the general functionality and runtime of XVSM, effectiveness, efficiency, and usability were the major evaluation criteria (chapter 6) regarding interaction and agility capabilities. The specific evaluation criteria and methods are described with respect to the concrete scenarios in the sections 6.1.1, 6.1.2, 6.1.3, and 6.2.

3.3. *Application Scenarios*

The XVSM framework is applied to three different application domains, namely the Traffic and Transportation domain, the Air Traffic Management domain, and the Production Automation domain to show the framework’s capability regarding coordination requirements and the ability to manage changing business requirements.

The following subsections describe the application scenarios and their special requirements.

3.3.1. Real-time, Safety-related Traffic Telematics (RealSafe)

The aim of the Real-time Safety-related Traffic Telematics (REALSAFE) project is to tackle the challenges for scalable and dependable vehicle-to-infrastructure (V2I) telematic systems in the Intelligent Transportation System (ITS) domain. V2I systems aim to make roads safer and their use more efficient. The deployment of services provides users with critical safety-related information, enabling a reduction in the number and severity of accidents. Additional value for future scenarios is that traffic can be more efficiently managed and congestions be avoided.

ITS is a decision support system that offers assistance in terms of instructions and recommendations to drivers, specially tailored to a particular dynamically changing context describing road and traffic conditions. Hence, its effectiveness depends on its ability to collect contextual data from many different sources (e.g., sensors, control centre) and appropriately generate and transport comprehensible, reliable and timely content to users. ITS are inherently complex in terms of the number of involved entities and interactions, including vehicles, on-board and road-side equipment, complementary communication technologies, producers and consumers of ITS data, distinguishable ITS services of different types. Therefore, ITS share challenges and requirements of pervasive/ubiquitous computing [49], e.g. scalability, context awareness, context management, heterogeneity, mobility, interoperability, security, adaptability and self-organization.

The RealSafe scenario assumes highways with fast moving vehicles communicating with a fixed, geographically distributed infrastructure, as illustrated in Figure 13. Along the highway there are so called Road Site Units (RSU) responsible for either passing safety and traffic information to the vehicles or receiving information from the vehicles and pass it to the system. RSUs exchange information via dedicated short range communication protocols (DSRC [237]) and are installed along the road network in 2-3 km distance to each other. They are connected by a meshed wired broadband network in order to assure scalability and increase fault-tolerance, Figure 14.

In the scenario the vehicle driver interacts directly with the vehicle's Application Unit (AU), which contains a dedicated Human-Machine Interface. Through a wireline connection within the vehicle, the AU is connected to the On-Board Unit (OBU). The OBU is responsible for maintaining the wireless link with the infrastructure, i.e. with the RSUs. Connectivity between the RSU and the passing by vehicles is characterized by a limited bandwidth, communication range, and connectivity window (ca. 300KB/sec for 2-3 sec at 100km/h in case of a single vehicle) allowing the exchange of small and a few messages only [238].

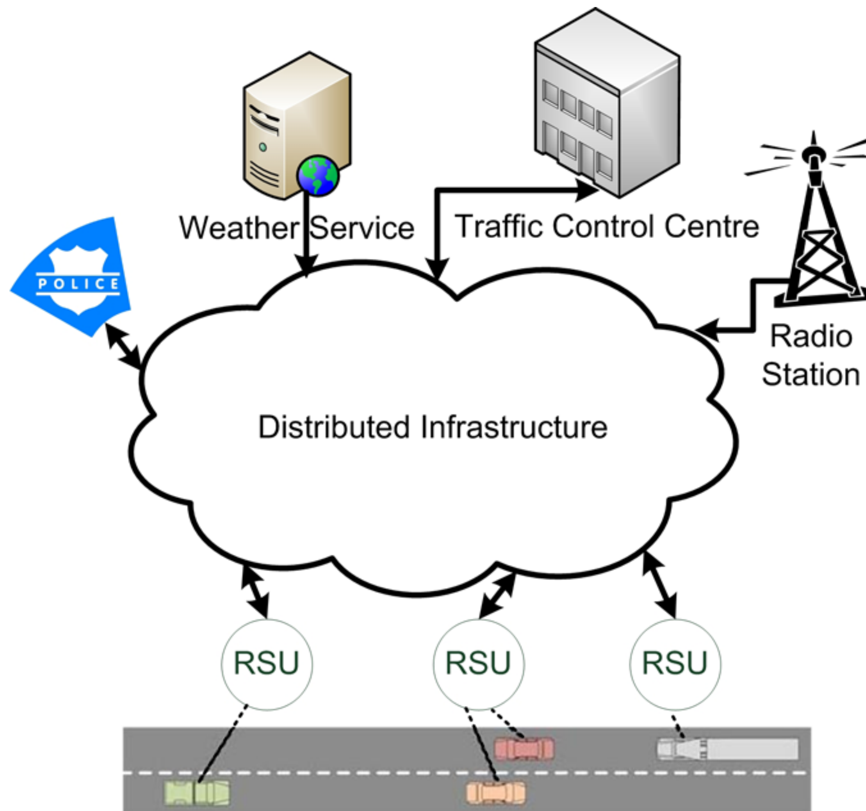

Figure 13: The structure of the RealSafe V2I System [125]

Exchanged messages are geo-located and their relevance in space and time is limited to a certain region, moving direction and period of time. Data belonging to a specific region needs to be queried and updated frequently as vehicles provide new information to the RSU and need the latest data from a RSU situated in the connectivity range. Therefore, the main objective of the infrastructure is to distribute these messages to relevant RSUs only. Depending on the type and severity of the message, the relevant RSU is one that is situated within a certain distance ahead (upstream) of the event (e.g. an accident).

Information exchanged in the system concerns the traffic (n.b. distribution of multi-media data is future work). Messages are published by e.g. the Traffic Control Centre (TCC), radio stations, the police, weather stations, the road maintenance depot, and of course the vehicles themselves. Messages exchanged or events generated may contain information about traffic restrictions and warnings (wrong-way-driver, speed limits, redirections,...), traffic density (the number of cars and their speed within a specific range,...), traffic congestions (location, length, duration, state updates,...), accidents (location, number of cars involved, blocked lanes, state updates,...), road conditions (wet, dry, temperature, number and location of road holes, humidity, hydroplaning warnings,...), current weather conditions (fog) and forecasts, or vehicle related information (acceleration statistics, break hits, sudden use of breaks, average and current speed, passed police control points, car condition, accident alert,...). The

published data is geo-located and as already mentioned its relevance in space and time is limited to a certain region, moving direction and period of time.

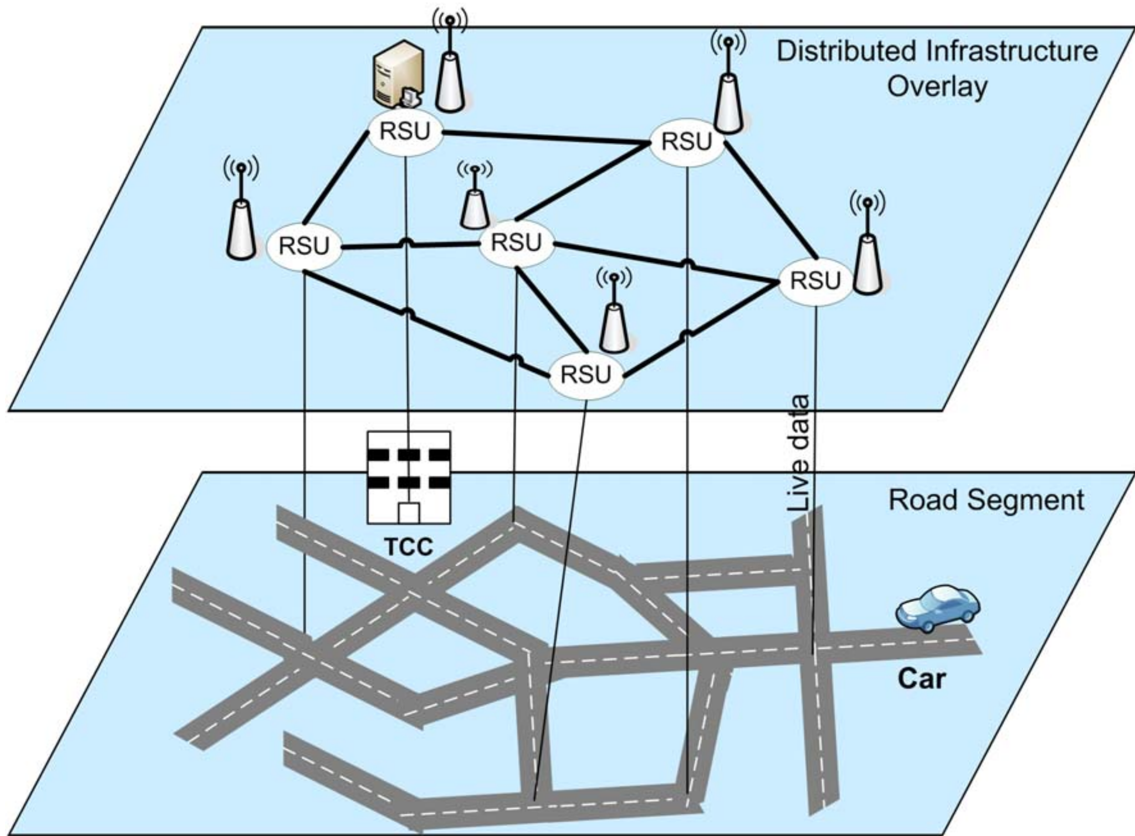

Figure 14: Distribution of a set of Road Site Units in a road-network and meshed communication network [125]

Receivers of exchanged messages are typically vehicles (driving at high speed) or road workers in field service. The received messages are used to generate statistics such as about the average speed/lane at forthcoming road segments, the number of vehicles/hour per direction in upcoming road segments, the distribution of vehicles over specific road segments, the average distance between vehicles, whether groups of vehicles decrease or increase their speed. This kind of information is used to adapt driving behavior since drivers are informed about occurrences and actions in upcoming road segments. Road workers in field use this information to prepare themselves and take precautions ahead in case of increasing traffic density. Road workers are also interested in statistics like the average temperature and the actual temperature curve of a specific road segment over a specific period of time. Summed up, receivers are interested in information which a) is represented by the very last event sent by the publisher, b) is represented by an aggregated set of events, or c) is a prioritized set of the delivered events. In a special case events can even cancel each other and should not be delivered at all. For example, a vehicle driver does not need to be informed about a wrong-way driver if that driver has already left the road.

However, the capabilities of this project are not only used to simply exchange information between the stakeholders, but to allow them (TCC, police, ambulance) coordinating each other in case of e.g., car accidents to efficiently provide health care for casualties.

3.3.2. System-wide Information Sharing (SWIS)

The aim of the SWIS project is to develop an “information sharing network” for the Air Traffic Management domain (Figure 15) which is characterized by very demanding safety- and security requirements and the need for high availability of services and network communication connections in an environment with heterogeneous middleware infrastructures, heterogeneous services, and changing business requirements.

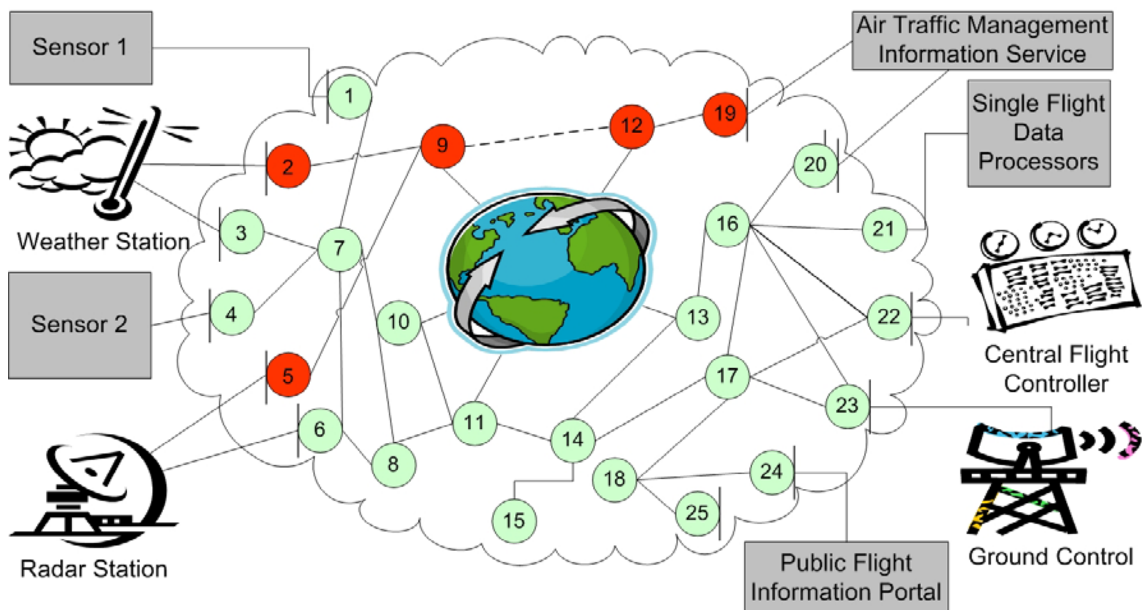

Figure 15: The complex world of the Air Traffic Management domain [155]

Figure 15 represents an example for the number of various services which need to be integrated over geographically large and distributed network groups in order to fulfill specific business goals. Main complexity factors refer to real-life scenarios consisting of over 8000 heterogeneous interconnected systems exchanging information based on more than 5000 different message types. Originally, services were integrated via point-to-point links and heterogeneity on network level was addressed by using so called adapters transforming messages between each used combination of middleware technologies. Thus, the need for integration of heterogeneous middleware technologies (Figure 16) with different APIs, transportation capabilities, or network architecture styles implies the development of static and therefore inflexible wrappers between each combination of middleware technologies, and thus increases the complexity of communication. Over the last several years, the message-oriented middleware [101] has

been used to connect applications in a loosely coupled manner but still requires low-level application coding intertwining integration and application logic. The resulting effort and complexity of implementing an integration platform with the support for any kind of existing middleware technologies and protocols therefore is considerably high. To enable transparent service integration, the Enterprise Service Bus (ESB) [40] provides the infrastructure services for message exchange and routing as the infrastructure for Service Oriented Architecture (SOA) [178]. It offers a distributed integration platform and clear separation of business logic and integration logic.

Figure 15 presents a set of services and the SWIS network consisting of intermediate network nodes connected by edges and forwarding messages to other nodes. There are two types of nodes: red nodes handle highly secure connections only, while green nodes do not provide specific security mechanisms. An edge refers to a network connection with specific characteristics, e.g. bandwidth. Services are listed on the left and on the right hand side. They are autonomous and loosely coupled, i.e., they do not know anything of each other apart from contracts providing and consuming data. Each service is connected to at least one network node. In the simplified example a business service is a sink service that requires a specific type of data to work properly; or a source service that produces data needed by sink services.

For instance, the business system Air Traffic Management Information Service (ATMIS) in Figure 16 has to provide information services about flights to business partners via a Public Flight Information Portal (PFIP). ATMIS needs to collect and refine information from at least 2 other systems: the Central Flight Controller (CFC) and the Single Flight Data Processors (SFDPs). As shown in Figure 16, the integration network consists of business services (e.g., CFC or ATMIS) which are connected to integration network nodes (e.g., CFC Node or ATMIS Node). Between these nodes, there exist different kinds of network links, represented by arrows between the nodes. These links use different transmission technologies (e.g., radio or wired transmission) as well as different middleware technologies (e.g., SONIC[8], TIBCO[9] or IPX-based middleware technologies [5]) for communication purposes. The capabilities of both kinds of technologies are explicitly modeled in order to automatically select suitable communication paths for particular service requirements, e.g., the red connection between (Figure 16) ATMIS Node, Node Y, and PFIP Node represents a reliable and secured communication path which is used by e.g., the ATMIS business service. The range of application types is wide. All of them have specific requirements with respect to reliability, timeliness, safety, failover, performance, auditability, maintainability, and flexibility which need to be fulfilled despite of heterogeneous environments.

---

[8] Progress SONIC[A] (http://www.sonicsoftware.com)

[9] TIBCO[A] (http://www.tibco.com)

[A] is a registered Trademarks and Commercial Off the-Shelf (COTS) product

Complexity is given due to the huge amount of possibilities how messages may be transported between services based on the properties of the network. In case the network consists of 17 nodes and 25 network links the number of possibilities how messages may be transported between 5 service interactions is about 17 billion. The challenge is to find the most suitable routing solutions within minimal time.

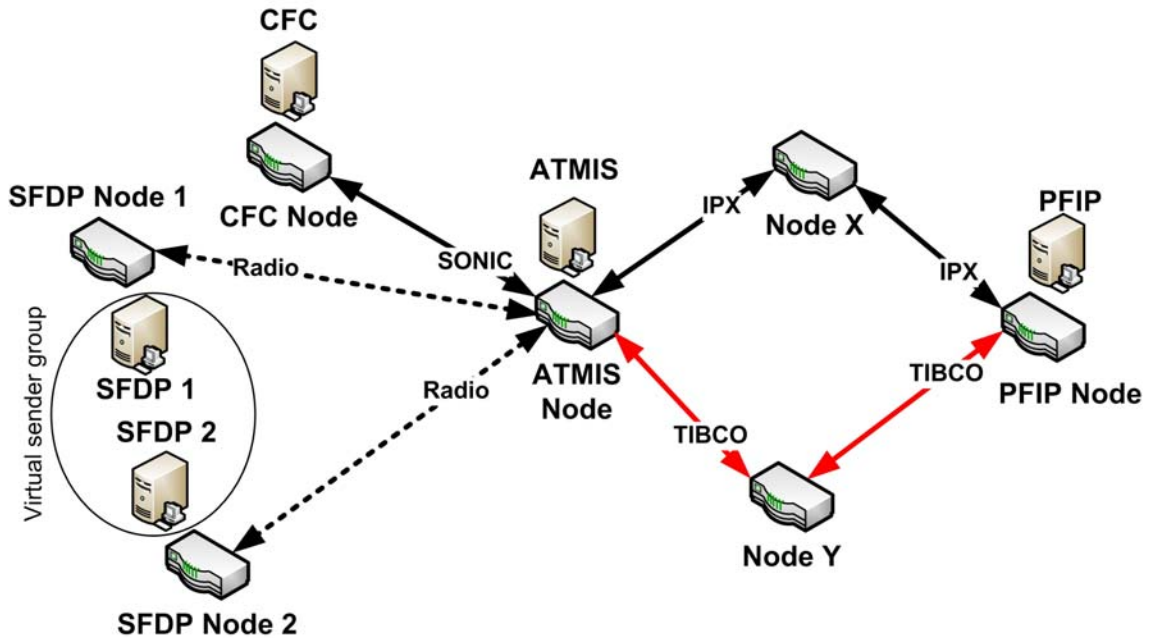

Figure 16: Heterogeneous network infrastructures and coordination requirements [152]

Beside different middleware technologies, the complexity of the SWIS project is increased by specific coordination requirements. On the left hand side of Figure 16, the services SFDP 1 and SFDP 2 are both elements of a so-called "virtual sender group". This means that they both send the same data, but only one message must be sent to the receiving ATMIS service. This is e.g. the case if the SFDP services are connected to redundant radar dishes, but only one message containing the radar data is needed; the other one is just produced for reliability purposes. So the services need to coordinate themselves in order to determine which SFDP service is allowed to send the message to the ATMIS service. In more advanced coordination scenarios, so called shadow nodes are required. Shadow nodes a) backup the main node in case it crashes, or b) verify calculations of the main node. In all these cases mechanisms are needed where e.g., the nodes themselves or the services running on them need to coordinate each other in order to fulfill business and infrastructure requirements.

3.3.3. Simulation of Assembly Workshops (SAW)

The SAW research project [126] investigates coordination requirements and recovery capabilities of software agents representing functional machines in an assembly

workshop. The overall goal is to increase the efficiency of the assembly workshop. This is achieved in two different ways, as described in the following.

The scenario from the production automation domain (Figure 17) consists of several different software agents each being responsible for the machine it represents. Such an agent may be:

- a **pallet agent (PA)** representing the transportation of a production part and knowing the next machine to be reached by the real pallet,
- a **crossing agent (CA)** routing pallets towards the right direction according to a routing table,
- a **conveyor belt agent (CBA)** transporting pallets, with optionally speed control, from one crossing agent to another,
- a **machine agent (MA)** controlling robots of a docking station for e.g., painting or assembling product parts,
- a **strategy agent (SA)** which, based on the current usage rate of the production system, knows where to delegate pallets, so that by taking some business requirements, like order situation, into consideration, a product is created in an efficient way, or
- a **facility agent (FA)** which specifies the point in time when machines have to be turned off for inspection.

The figure shows a software simulator for a production system, to be more precise for an assembly workshop. Such manufacturing systems are very complex and distributed. The usage of a digital simulator instead of a miniature hardware model has a lot of advantages like, low operating costs, the easy reconfiguration or parallel testing.

Multi-agent system (MAS) [235] is an accepted paradigm in safety-critical systems, like the production automation. A major challenge in production automation is the need to become more flexible. The requirement is to react quickly to changing business and market needs by efficiently switching to new production strategies and thus supporting the production of new market relevant products. However, the overall behavior of the many elements in a production automation system with distributed control can get hard to predict as these elements may interact in complex ways (e.g., timing of fault-tolerant transport system and machines) [137]. Therefore, an issue in this context refers to the implementation of agents with reduced complexity of their implementation by e.g., minimizing the number of interactions.

Another key requirement to improve the efficiency of production systems is to minimize downtime of the production system in order to allow just-in-time delivery of business claims and minimal production costs. Nevertheless, as any other distributed system, MAS are prone to failures as well, and consequently an agent may crash. Therefore, the second aim is to reintegrate crashed agents into the production system as

fast as possible by reducing the time it needs for recovery, i.e. the time needed to collect information about the environment to catch up with the other agents. Thus, the production system shall become capable of operating at optimal level with zero-delay.

Figure 17: View of a simulated Production Automation System[10] [124, 224, 225]

3.3.4. Summary

Summarizing the previous sections, all scenarios do have the following issues in common a) the need for efficient coordination and data exchange, both in centralized and distributed environments of a large number of components b) being able to cope with changing requirements and infrastructures and to adapt quickly to the new circumstances, and c) reducing the complexity in applications coming along with distributed systems.

[10] Thanks to Rockwell Automation for the provision of the simulator.

Chapter 4

4. The Space-based Computing Paradigm

This chapter summarizes the proposed Space-Based Computing (SBC) architectural style. In the first section, an overview of the architectural style is given. The second section presents XVSM (eXtensible Virtual Shared Memory) as reference architecture for the realization of the SBC architectural style. The section describes its architectural concepts of containers, coordinators, selectors, and aspects. It illustrates in detail how various coordination and changing business and infrastructural requirements can be realized. The third section explains how traditional architectural styles can be implemented by means of XVSM.

4.1. *SBC Overview*

Similar to the Linda coordination model, SBC is mainly a data-driven coordination model, but can be adapted and used according to control-driven coordination models as well (section 4.3). As shown in Figure 18, application components running on different physical nodes coordinate each other by means of writing, reading, and removing (section 4.2.1) shared structured entries from a logically central space entity. SBC is a logical central architectural component and does not specify where it is physically located (Figure 20, part 5).

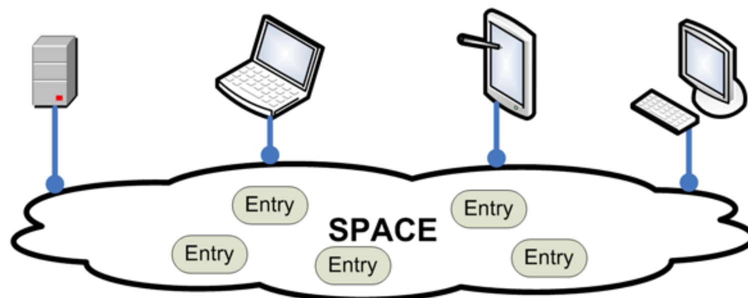

Figure 18: High-level view of the Space-Based Computing Paradigm

An implementation of the SBC architectural style can be deployed on a physical central server, or run on several nodes (Figure 19 and Figure 20, part 5). In the latter case, internal mechanisms have to make sure, that the shared data structures on the participating nodes are synchronized by taking into account use case specific requirements (section 5.1).

For instance, a strategy may define that all nodes have to host the same consistent data set. From the application component's point of view there is no client/server or P2P architectural style to worry about. The application component has a specific role in a coordination policy (section 4.1.1) the component is part of and acts according to which has been defined explicitly by higher business goals or autonomously. The used coordination policy specifies the component's role, whether it is a client requesting information or a server responding with data, or both. In the following, this thesis refers to the term "space" in the same way as to SBC. In the next sections, SBC's coordination and agility capabilities are going to be described briefly by means of Figure 20, and in more detail in section 4.2.4.

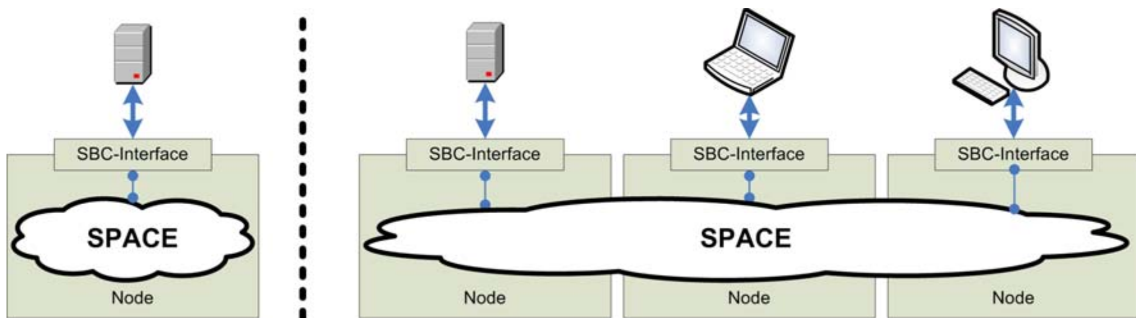

Figure 19: Examples for client/server (left) and distributed architectures (right) for a space

4.1.1. Coordination Policies

Distributed application components (Figure 20, part 1) access the space by means of *read*, *take*, and *destroy* operations if they want to retrieve and/or remove data, or by means of *write* operations in case of adding data (section 4.2.1). The *notify* operation allows application components to be notified about changes in the space.

Data placed in the space is stored in a structured way according to exchangeable coordination policies (Figure 20, part 3). Each policy is responsible to structure and maintain data in the space. Several coordination policies may exist at the same time, thus the same data may be structured differently, depending on the policy looking at. In its simplest form, such a policy is similar to the Tuple Space coordination model that uses template matching on structured tuples. Other policies may involve FIFO, LIFO, keys, geo-coordinates, or even more complex coordination like the market place pattern. Writing data using the last policy could mean e.g., to announce a new offer about which other application components are notified in near-time.

The efficiency of coordinating application components mainly rests upon the efficiency of the coordination policy. Internal mechanisms have to allow access of multiple concurrent operations while preserving consistency. The way of managing synchronizations and their granularity depends on the represented policy and therefore requires careful trade-off analyzes.

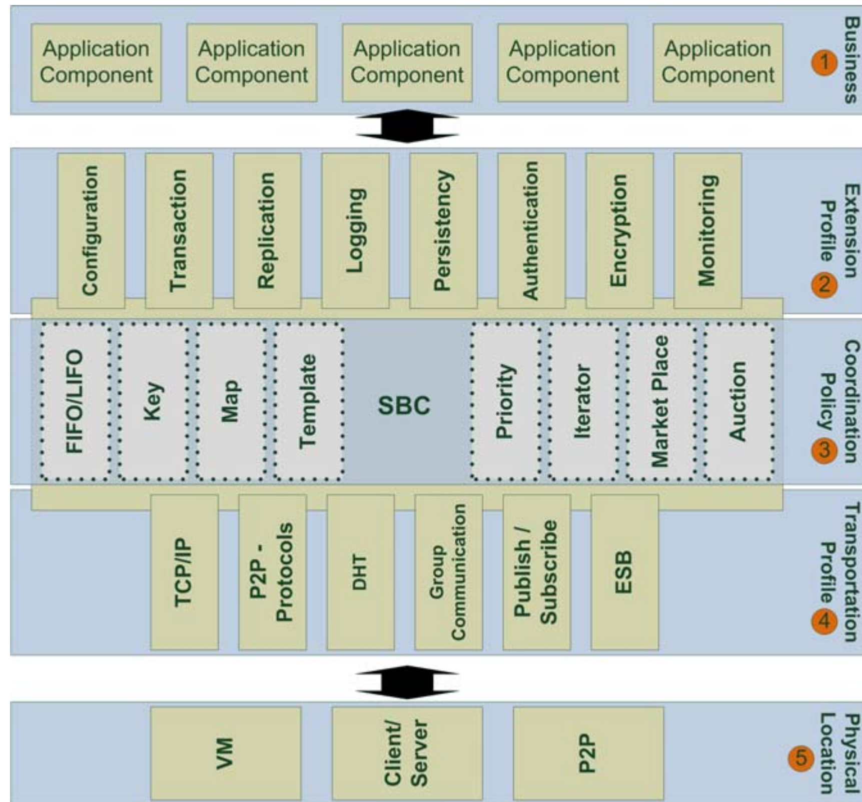

Figure 20: Overview of the Space-Based Computing Architectural Style

4.1.2. Profiles Enabling Agility

Two classes of profiles can be dynamically injected into the architecture: transportation and extension profiles.

Transportation profiles implement mechanisms (Figure 20, part 4) for transporting data between nodes. The actual implementation of a transportation strategy is secondary and depends entirely on business requirements or technical constraints. The profile is responsible for transmitting data between:

- application components and the space to execute their operations on that space
- several distributed spaces to enable inter-space communication for e.g., synchronization purposes

Extension profiles (Figure 20, part 2) realize non-functional strategies, like replication or notification mechanisms, needed to operate the space efficiently. Extension profiles are responsible to extend the capabilities of the space in order to provide more capabilities to the coordinating application components. Addition and deletion of profiles can be performed transparent to the application components allowing the platform to change its properties without affecting the application components.

4.2. *XVSM- eXtensible Virtual Shared Memory Architecture*

Since an architectural style is very general, it is necessary to introduce its capabilities by means of a reference architecture called eXtensible Virtual Shared Memory Architecture (XVSM). The XVSM reference architecture has been implemented in three different programming languages resulting in several prototype implementations. It has been realized in Java resulting in the prototype implementation called MozartSpaces [186, 197], in .Net resulting in the prototypes called XCoSpaces [113, 196] and TinySpaces [141], and in Haskell [50]. Figure 21 shows relations between the various terminologies. On the top there is the SBC architectural style that has been designed by means of the concepts collected in the XVSM reference architecture. The reference architecture is implemented by the previously mentioned prototypes. XVSM is just an example how the SBC architectural style can be realized. There are also other ways to implement the concepts.

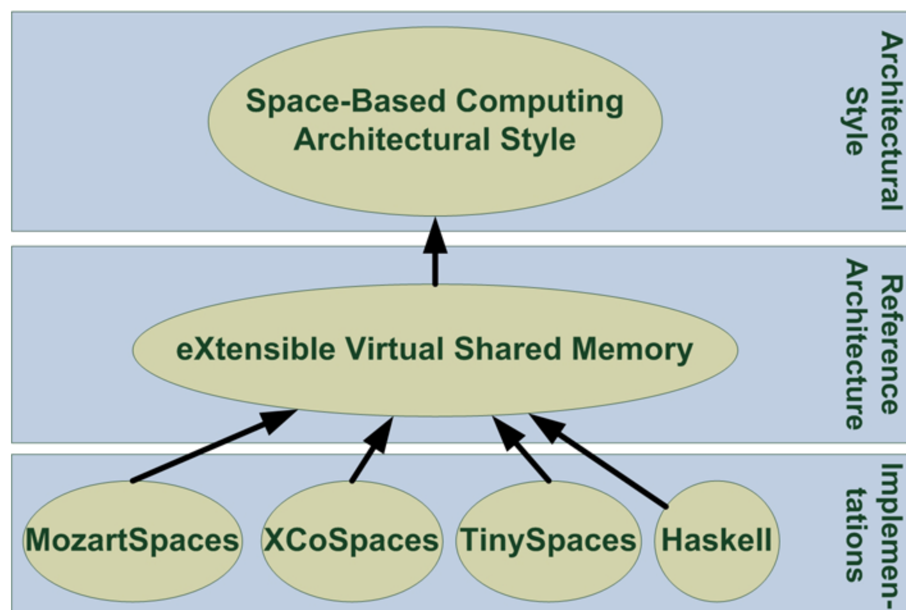

Figure 21: XVSM and its various implementations

The following sections picture the features and components of the XVSM reference architecture based on the latest MozartSpaces implementation in detail. Figure 22 illustrates a general overview of the XVSM reference architecture divided into an application part (left side) and a space part (right side). In the following the sections describe the components of the right part first, including the sub-components coordinators, containers, and aspects. Section 4.2.1 details the components' characteristics, their interfaces, and the semantics of supported operations. Section 4.2.2 describes the execution of operations.

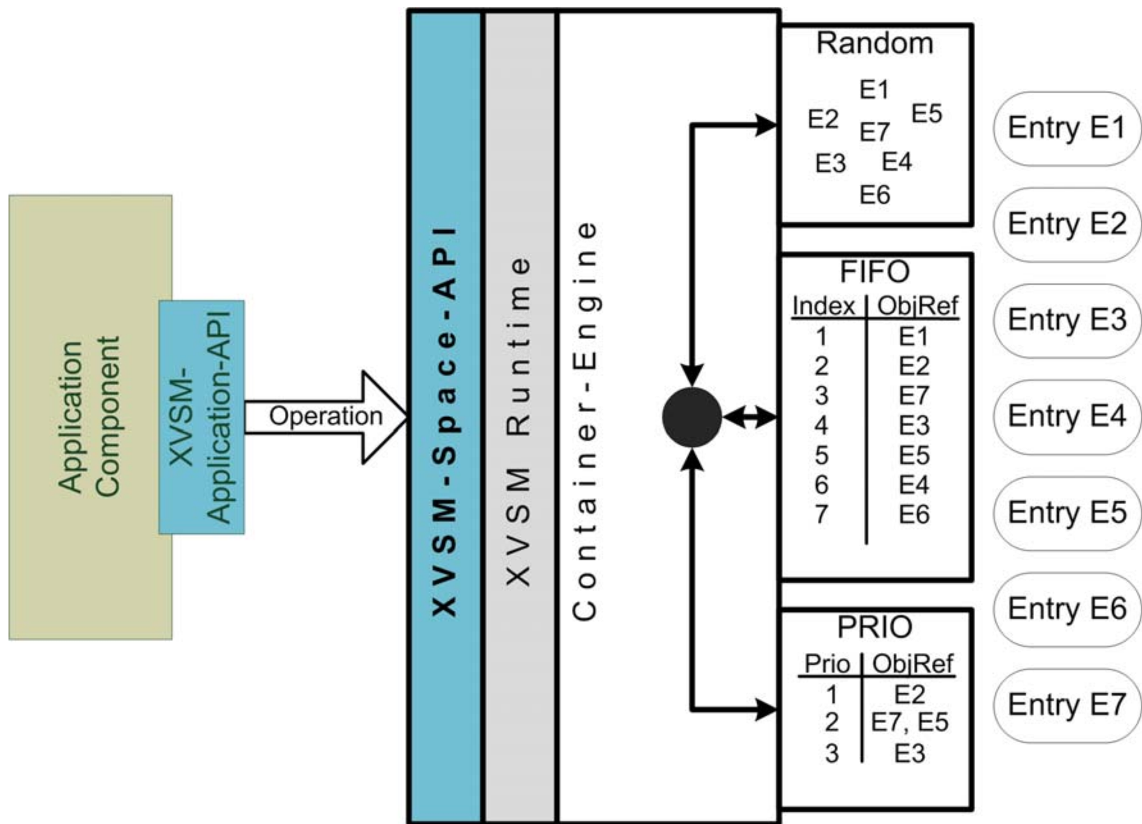

Figure 22: XVSM architecture with a container hosting a random-, a FIFO-, and a PRIO coordinator structuring 7 entries [124]

4.2.1. Container-Engine

As in Linda, in SBC application components coordinate each other by means of placing and retrieving data into/from a shared "space". In XVSM data is stored in so called containers that can be interpreted as a bag containing data entries. In XVSM multiple containers may exist at the same time and the number of containers defines the XVSM space. If the number of containers equals zero, then the space is called "empty". The responsibility of the container-engine layer is the creation and destruction of containers.

In its basic form a container is similar to a tuple space - a collection of entries. The main difference to a tuple space is that a container

- extends the original Linda API with a *destroy* method
- introduces so called coordinators enabling a structuring of the space
- may be bounded to a maximum number of entries

Data items stored in a container are called "entries". In MozartSpaces an entry is either of type Tuple or of type AtomicEntry. A Tuple contains other entries, which in turn are either AtomicEntries or other Tuples. An AtomicEntry is a Generic Java class.

4.2.1.1. Container API

As in Linda (*out*, *in*, *rd*), a container's interface provides a simple API for *reading*, *taking*, and *writing* entries, but extends the original Linda API with a *destroy* operation. Similar to a *take* operation, a *destroy* operation removes an entry from the container. Although a *destroy* operation could be mapped onto a *take* operation where the result is omitted, it is still necessary to induct this kind of operation that does not return an operation value. The reason is that this way a lot of data traffic and execution time is avoided since the removed data does not need to be transferred back to the initiator of the operation.

The *destroy* operation is also helpful especially in the case of bulk operations [96]. Containers support bulk operations, so that it is possible to insert multiple entries into a container resp. to retrieve/remove multiple entries out of it within one operation.

While Linda makes an explicit distinction between blocking (*rd*, *in*) and non-blocking (*rdp*, *inp*) primitives, XVSM primitives are restricted to the four basic operations. Whether an operation blocks depends on the coordination policy a coordinator represents.

4.2.1.2. Coordinator

A container possesses one or multiple coordinators. Coordinators implement and are the programmable part of a container. They are responsible for managing certain views on the entries in the container. The aim of a coordinator is to represent a coordination policy. Each coordinator has its own internal data structures which help it perform its task. If the business coordination context and requirements are known beforehand, the coordinator can be implemented in an efficient way with respect to its policy. Any number of coordinators can be added to a container, but a container hosts at least a system coordinator that is added automatically and implements a non-deterministic behavior.

A coordination policy is represented in the implementation of each coordinator. Although the semantics of two coordinators may be the same, they may be implemented in different ways if taking into account business specific requirements. This way a coordinator has an optimized view on the stored entries by taking into account scenario specific coordination requirements. An example is given in [115] where a Linda coordinator (i.e. a coordinator that represents Linda tuple space behavior) has been optimized for read operations resulting in O(logN) effort.

Figure 22 shows three exemplary coordinators (Random, FIFO, and PRIO Coordinator) referencing seven entries (E1-E7). In the MozartSpaces implementation each coordinator uses Java object references for managing its view on the stored entries. The

Random Coordinator contains the references to all existing entries in the container and returns/removes an arbitrary entry in case of *read/take, destroy* operations. The FIFO Coordinator imitates a queue. It stores in the lowest index the reference to the entry that has been in the container for the longest time and in the highest index the reference to the entry that has been added last. The PRIO Coordinator is a so called optional coordinator (section 4.2.1.4) and groups references only to specific entries according to a priority defined by the software developer.

In general, whenever an operation is performed on a container, the parameters of the operation are collected in a so called selector. Every coordinator has its specific selector which can be interpreted as the coordinator's logical interface for the performed operation. Comparing the relation between selector and coordinator with OOP concepts, the selector is the interface and the coordinator the actual implementation of that interface. In case of *read*, *take*, and *destroy* operations the selector contains parameters (like a counter for the exact number of entries to be retrieved) for querying the view on the managed entries. In case of a *write* operation parameters influencing the coordinator in updating its view are required.

For instance, the selector of a Linda coordinator contains the template to be used when searching for entries matching that template. However, the FIFO and Random coordinator do not need any selectors since the semantics of their coordination policy already define how to find the entry. In case of a Key coordinator (comparable to a hash in Java) the *write* operation requires the key that should reference the written entry in the coordinators view.

In case a container hosts several coordinators, operations may define multiple selectors as well. The number of specified selectors depends on the business coordination requirements and is not bound to the number of coordinators in the container. If more than one selector is used in querying operations, the outcome of the execution of the first selector will be used as input to the second and so on [128]. For example, if a FIFO Selector with count 10 is used together with a Label Selector with value "X", the container engine asks the FIFO Coordinator to select the ten first entries and afterwards it asks the Label Coordinator to look whether one out of these ten entries is referenced by the label "X".

4.2.1.3. Bounded Container

A bounded container is a container where independent of the deployed coordinators the number of entries is limited to a number greater than zero. When the number of entries has reached the maximum permitted number of entries, then the container is called "full".

4.2.1.4. Execution of Operations

The container-engine has to make sure that the execution of the operation between participating coordinators in case multiple selectors are defined is done correctly. The sequence of selectors in read operations is non-commutative *AND* concatenated (i.e. filter style). This means that it makes a crucial difference if 10 entries are selected from a FIFO Coordinator and then a template matching is performed or if the template matching is done first and then the FIFO Coordinator tries to return ten entries.

Before explaining how operations are executed two classes of coordinators have to be introduced. The software developer may declare a coordinator at the time of its creation to be either obligatory or optional. An obligatory coordinator must be called for every write operation on the container, so that a coordinator always has a complete view of all entries. An optional coordinator, however, only manages entries if it is explicitly addressed in the write operation, while other entries in the container remain invisible. The FIFO Coordinator can be used as an obligatory one since it does not need any additional parameters.

In the following the execution of the operations in the container-engine is explained in general. The given explanation does not contain any information about the different semantics of XVSM operation transactions or operation timeout. Those aspects are described in [50] in detail.

Write

First, the *write* operation is executed on all optional coordinators for which parameters have been specified. Afterwards, the *write* operation is executed on all remaining obligatory coordinators even if the operation cannot provide parameters for those coordinators. When a *write* operation has to be blocked depends on the semantics of the coordinator. A semantic may be that an operation has to block if for instance, a Key Coordinator already has a key in its view that the *write* operation of a new entry uses too.

Read

The container-engine iterates over the specified selectors of the operation and queries the corresponding coordinators. In case multiple selectors are specified the result set of the first queried coordinator is the set the next coordinator has to use to execute its query. A *read* operation has to be blocked in case the query cannot be satisfied.

Take

A *take* operation is executed the same way as a *read* operation whereas the result set of the last coordinator defines the set of entries which have to be removed from the container. Therefore, before returning the result set to the initiator of the operation the

container-engine asks all coordinators which store a reference on the entries of that result set to remove the entry from their views. Similar to a *read* operation, a *take* operation has to be blocked in case the query cannot be satisfied.

Destroy

A *destroy* operation is executed like a *take* operation without returning the final result set to the initiator of the operation. Similar, a *destroy* operation has to be blocked in case the query cannot be satisfied.

4.2.2. XVSM Runtime

The XVSM Runtime is a layer that is responsible for executing the basic operations by concurrent runtime threads. Operations executed in the container-engine are called requests in the XVSM Runtime layer. Beside the operation itself requests contain context specific meta-information (e.g., timeout, location of the receiver of the request result). Since Linda primitives block if a tuple does not match a specific template, the XVSM Runtime is also responsible for managing the blocking semantics of operations. The difference to the Linda coordination model is that XVSM Runtime can alter the semantics of the initiated request. This is achieved by so called aspects. The following sections explain how aspects work and how requests in general are managed.

4.2.2.1. Aspects

The XVSM Runtime layer realizes some parts of Aspect-oriented Programming (AOP) [116] by registering so called aspects at different points before the operation accesses the container-engine or when the operation returns from the container-engine. Aspects are executed on the node where the container is located and are triggered by operations either on a specific container or on operations related to the entire set of containers, the space. The join points of AOP are called interception points (IPoints). Interception points on container operations are referred to as local IPoints, whereas interception points on space operations are called global IPoints. IPoints are located before or after the execution of an operation, indicating two categories: pre and post. Local pre- and post-IPoints exist for read, take, destroy, write, local aspect appending, and local aspect removing. The following global pre- and post-IPoints exist: transaction creation, transaction commit, transaction rollback, container creation, container destruction [50].

For each request a so called Aspect Context with its parameters is passed along allowing applications and aspects to influence (i.e. parameterize) the operation of installed aspects. In case multiple aspects are installed on the same container, they are

executed in the order they were added. Adding and removing aspects can be performed at any time during runtime.

In general, Figure 23 shows a container with three local pre and three post aspects and their various return values. The XVSM Runtime layer accepts incoming requests and passes it immediately to the first pre-Aspect of the targeted container. The request passed to and analyzed by the aspect contains the parameters of the operation, like entries, transactions [196], selectors, operation timeout, and the Aspect Context. The called aspect contains functionality that can either verify or log the current operation, or initiate external operations to other containers or third-party services. Aspects can be used to realize security (authorization and authentication), the implementation of highly customizable notification mechanisms, or the manipulation of already stored or incoming entries.

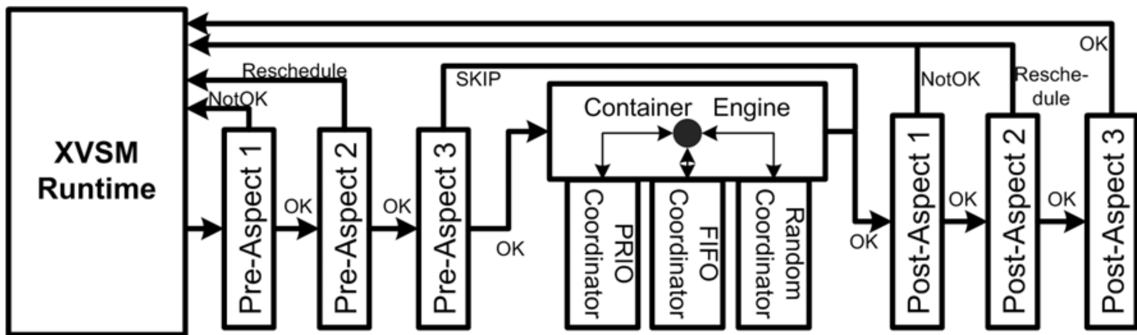

Figure 23: Execution sequence and return values of Aspects and data- and control-flow in a container with three installed pre- and post-Aspects [125]

The central part of a container is the implementation of the container's business logic, i.e. the storage of the entries and the management of coordinators. A request is successful if it passed all pre-aspects, the container-engine, and all post-aspects without any errors. However, an aspect may return several values by which the execution of the request can be manipulated. The following return values are supported:

- OK: The execution of the current aspect has been finished and the execution of the next aspect or of the operation on the container proceeds.
- NotOK: The execution of the request is stopped and the transaction is rolled back. This can be used by e.g. a security aspect denying an operation if the user does not have adequate access rights.
- SKIP: This return value is only supported for pre-aspects and triggers the execution of the first post-aspect. This means that neither any other pre-aspects nor the operation on the container is executed.
- Reschedule: The execution of the request is stopped and will be rescheduled at a later time. This can be used to delay the execution of a request until an external event occurs.

Depending on the result of the last post-aspect the result of the request is either returned to the initiator of the request, or the request is rolled back.

4.2.2.2. Virtual Container

Based on the mechanisms aspects provide, a special type of container is introduced facilitating the transparent access to third-party resources. A Virtual Container is a container that consists of at least one pre-aspect that skips every incoming operation (Figure 24). The container itself may keep several coordinators, but in such a case they are not executed, since the aspect skips the operation on the container engine. A Virtual Container can be used for operations without the need for data storage, or to access third-party technologies. In both cases the container represents resources rather than a coordination model. Such a resource may be a database that is accessed by one of the aspects.

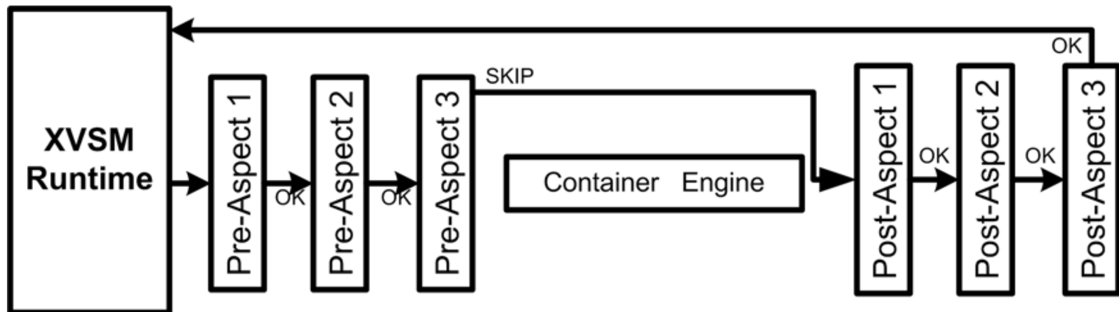

Figure 24: The concept of a Virtual Container

4.2.2.3. Managing Operations

Figure 25 shows the architecture of the XVSM Runtime layer [50, 196]. Beside the already mentioned container-engine and pre- and post-Aspects, the architecture consists of the XVSM-Space-API, the Event Processor, the Timeout Processor, and of the Request, Response, and Wait Containers.

The XVSM-Space API extends the container-engine API by the methods *add* and *remove* aspect. The implementation of the API accepts incoming requests and places them immediately into the so called Request Container that can be seen as a request storage component. Requests are received from either from another XVSM-Space-API or from a XVSM-Application API (section 4.2.3) representing a programming language specific binding to the space [196].

Runtime threads execute requests from the Request Container concurrently. The request is taken and "passed along" pre-aspects, container-engine, and post-aspects. Finally, the result of the request is placed into the Response Container. Once the request has passed

post-aspects a copy of that request is sent to the Event Processor. If the request indicates that the operation has to be blocked the Event Processor writes that request into the Wait Container. The Wait Container stores blocking operations. If there is no such indication then the Event Processor knows that the container has been changed, and probably a blocking operation can be executed. Therefore, the Event Processor searches for a request that matches the executed operation. For instance, if a take or destroy operation was performed then it means the container is one entry short indicating that one of the blocking write operations may be successfully executed this time. Therefore the Event Processor takes all write requests with regard to the effected container and places them into the Request Container. On the other hand, if a write operation was executed successfully then the Event Processor places all read, take, and destroy requests with regard to the effected container into the Request Container.

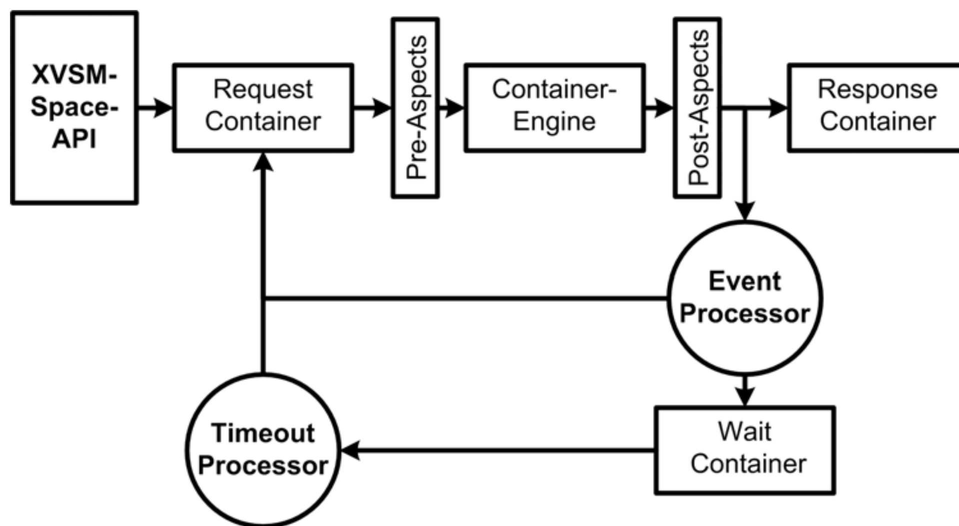

Figure 25: Architecture of the XVSM Runtime Layer

In XVSM every request that can block contains an optional timeout parameter, which represents the maximum time period which the operation can be (re-)executed. An operation with timeout 0 is executed exactly once and in case it cannot be fulfilled it will not block. An operation with infinite timeout will wait until it can be fulfilled successfully. The Timeout Processor supervises the blocked requests in the Wait Container. It removes expired requests from there and places them into the Request Container. In the course of executing the request the runtime thread realizes that the request has already expired, ignores the request, and places a response with the appropriate error into the Response Container.

The Response Container stores responses containing the result of the request or an error message. Runtime threads remove the response from the container and send them to the location specified in the original request. The location is either another container accessed via the XVSM-Space API or the requesting application component reachable via the implementation of an XVSM-Application API.

4.2.2.4. Reachability of Containers

Containers are Internet addressable using an URI of the addressing scheme "*xvsm://namespace/ContainerName*", like “xvsm://host.mydomain.com:1234/CName”. Every XVSM Runtime hosts several different transportation profiles responsible for accepting requests and sending responses over the physical network. Accepted requests are placed into the Request Container. Responses taken from the Response Container are sent to the URI specified in the context of the response (i.e. this is the same parameter as the context of a request).

The protocol type "*xvsm*" makes the usage of transportation profiles transparent to the application component. This means that as a transportation medium for accessing that particular container one of the transportation profiles is used without impact on the application component. The application component may specify the properties of transportation (e.g., reliability). XVSM Runtime uses these properties to search for the profile that is capable of supporting these requirements. Depending on the application domain, or on the underlying network infrastructure, "*xvsm*" may be translated to e.g. *java-via-tcp*, or a P2P protocol.

4.2.3. XVSM-Application API

The XVSM-Application API extends the XVSM-Space API with a notify method. It is a programming language specific implementation which communicates with the XVSM-Space API via a so called XVSMP protocol [50]. The protocol represents XVSM-Space API requests mapped on an xml structure. The exchange of requests via XVSMP over a transportation profile is performed in an asynchronous way.

This mode of communication between distributed application components is important in comparison to synchronous mode of communication, since assumptions like reliable network, zero latency, or homogeneous networks do not hold true in distributed environments [60]. With respect to these assumptions, the Answer Container is introduced that allows the application component to continue processing and to retrieve the result of the submitted operation whenever it fits best for the application.

4.2.3.1. Answer Container

Whenever the application component invokes a method, the request contains the identifier of the answer container as argument to the called operation where the result to that request is placed by the runtime threads [51].

Figure 26 shows the sequence of processing in case the application component has provided an answer container. The operation is executed on the targeted container,

however, instead of returning the result to the application component, the result is written into the answer container. An answer container is either a physical or a virtual one.

A physical answer container is an ordinary container as described in section 4.2.1. Deployed aspects on that container may provide additional use case specific functionalities, like the aggregation of incoming results. It is not bound to a single application component and thus can be shared among several application components to e.g., enable load balancing properties. The response stored in the answer container is retrieved by either polling or notification mechanisms (section 4.2.3.2) as shown in Figure 26.

A virtual answer container is addressed the same way as a physical one, but represents a binding in the XVSM-Application API between the identifier of the answer container and a callback method provided by the application component. This means that whenever an entry is written into a virtual answer container, that entry is forwarded to a specified callback method thus the result to a request is pushed to the application component.

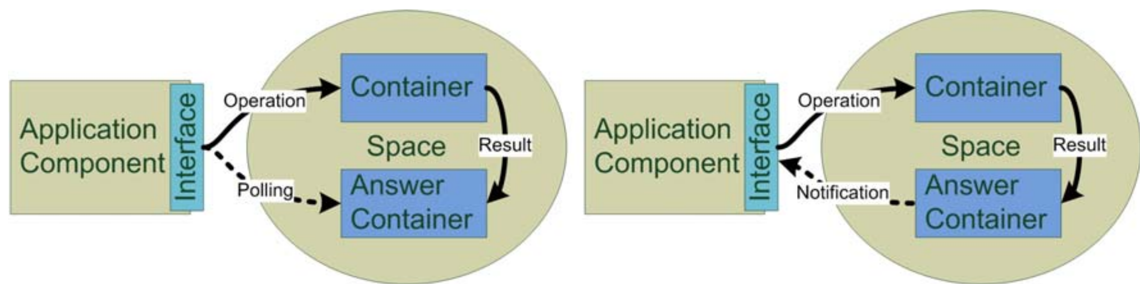

Figure 26: The concept of Answer Containers

It has to be mentioned that in MozartSpaces it is also possible to invoke methods in a synchronous manner. Nevertheless, even if an application component calls a method in a synchronous manner, every operation is executed in an asynchronous way. The reason is that the interface dispatches the method call asynchronously and redirects the calling process to wait for an entry in a specified virtual answer container. Therefore, from the application component's point of view the process is still synchronous, although the execution itself is performed asynchronously.

4.2.3.2. Notification

Figure 27 shows the general structure of processing a notification in XVSM. In the example there is a container "X" and an application component 3 that wants to be notified whenever container X is accessed. When that application component invokes the notify method, XVSM Runtime registers an aspect (e.g., a so called notification aspect) on container X and creates a so called notification container. The notification

aspect intercepts the processing of the operation on container X and writes data into the notification container. When the operation is intercepted (pre or post) information (e.g., a copy of the executed operation or use case specific information about the operation) is written into the notification container, depending on the scenario. A notification container is a container as described in section 4.2.1, thus it is capable of hosting additional pre and post-aspects for e.g., aggregation of entries.

Beside the notification aspect and the notification container, XVSM Runtime performs *take* operations on the notification container and specifies a virtual answer container where the result for that *take* operation has to be placed. The virtual answer container is bound to a call back method of the application component, thus whenever an entry is written into the virtual answer container the application component receives that entry. This way an application is notified about events on container X.

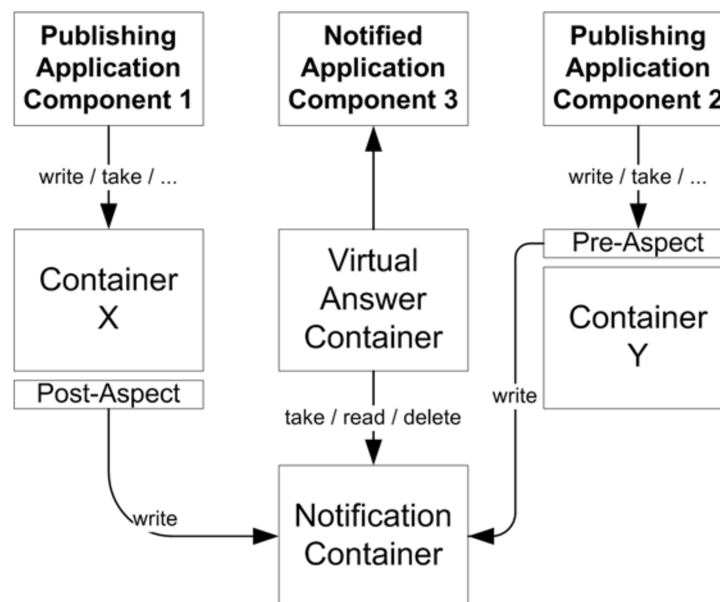

Figure 27: General structure of an XVSM Notification [127]

As it can be seen, the introduced notification mechanism builds on already described XVSM architectural concepts. This allows software developer to create domain and application specific notification mechanisms which exactly meet given requirements. The described mechanism shows several points where tuning of notification is possible, as illustrated in the following:

The notification aspect can be placed either before or after the execution of the operation on the container. If the aspect is installed as a pre-aspect, the application component cannot be sure whether the operation was really successfully executed on the container or had to be aborted due to errors. Furthermore, if the operation has to be blocked the application component is notified every time that operation comes to execution. The notification aspect can be registered for any XVSM operation (read, write …) and therefore a notification cannot only be created when an entry is written. It

is also possible to create notifications which notify a user when entries are read, taken or deleted.

The notification aspect is an aspect as described in section 4.2.2.1 (it receives all information about the request) and writes entries into the notification container. The notification aspect writes either entries containing the operation carried out or any other information (i.e. filtered, enriched with information or modified) application component 3 wants to be notified about. For example a notification can be triggered if a specific order of messages has been detected, or if the aspect context of the request matches specific values. It is also possible to notify the subscriber about meta-information instead of the content. For example a monitoring application component could be interested in the memory consumption of the XVSM instance after each operation. In this case the notification aspect would completely ignore the operation. It would read the current memory usage of the system, store this information in an entry, and write this entry into the notification container.

Figure 27 does not define where the shown containers are placed physically. It is possible that the containers are on the same node or on different ones. The latter one enables the creation of durable subscriptions [68] by placing the notification container on a node which is always reachable. The notification events are collected in the notification container whether or not the client is reachable. When the subscribing application component is online again, the XVSM Runtime fetches the notifications from the notification container which contains new entries written during its absence and pushes them via the specified call back method to the application component.

The operation which is used to get the information from the notification container can be varied. When a subscriber is not interested in the content of the notification but in the fact that something has happened, a delete operation is preferred. When the delete operation with the given selectors returns (i.e. an answer is written to the answer container) the subscriber knows that a matching event occurred. Furthermore, the notification container may host different coordinators. For example, if the order in which operations occur is important a FIFO coordinator could be. This ensures that even when the subscriber is offline the notifications are kept and remain in the correct order.

Finally, the notification container is not restricted to be written by one notification aspect only. If application component 3 is interested in a merged notification regarding several different containers, then several notification aspects may write appropriate entries into a single notification container.

4.2.4. Supported Ways of Decoupling

The main concepts of the XVSM architecture mentioned in the previous sections, allow to define a structuring of concepts to separate concerns and thus to allow the software developer of distributed applications to cope efficiently with complexity issues. The concepts can be explicitly clustered and categorized resulting in models distinguished by their capabilities for managing computation, coordination, organization, distribution, and communication requirements. In the following sections the identified categories are described in detail and shown in Figure 28.

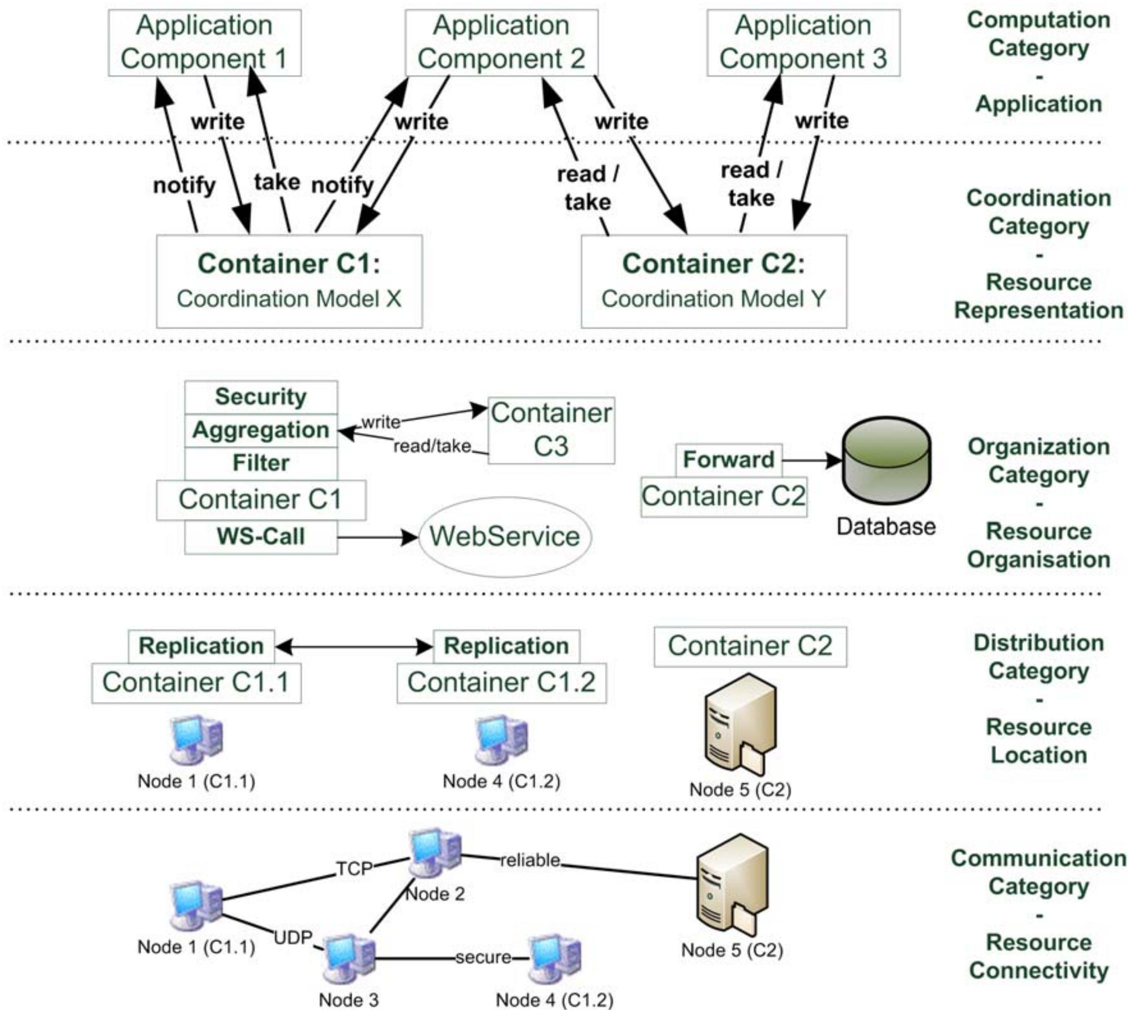

Figure 28: The five categories of decoupling in XVSM [151]

4.2.4.1. Computation Category

This category represents the application logic. The main point at this category is to specify what the application has to do with the data received and when to write new data into which containers. The application has to know how to access containers. For

the application a container may look like an endpoint known from ESBs [40]. Endpoints abstract underlying protocols trying to map high-level process flows into individual service invocations. A container abstracts transportation protocols as well, but in addition offers the capability to use other coordination models beside the FIFO coordination style, that represents a simple message queue used in ESB. This means, that although the container contains entries - written by some applications - they may still not be accessible according to the coordination model.

4.2.4.2. Coordination Category

As stated in section 2.2.2.2 the computation model is used to express the computation requirements of an algorithm, while the coordination model is used to express the communication and synchronization requirements. This decoupling allows e.g., a fifo style of coordination to be switched to e.g., lifo style of coordination, or more complex coordination models transparent to the application accessing the container. This is done by replacing the existing coordinator of the container.

However, care must be taken if business coordination requirements change which may trigger a switch to another coordinator. This may result in further changes in the application component. If the selector of the new coordinator is a different one then the implementations where XVSM primitives are used have to be updated. If the semantics of business coordination (e.g., blocking behavior) – and thus of the coordinator - has changed, aspects may help mapping between the semantics of the old and of the new coordinator to some extent, thus diminishing changes in the application component.

4.2.4.3. Organization Category

The organization category hides from application components the various architectural styles, manages cross-cutting concerns like security and interconnections between containers, and abstracts third-party resources. Architectural styles explicitly designed for coordination issues, like the blackboard style (see section 2.3.3.2), have already been taken into consideration in the coordination category. Since XVSM requests are handled asynchronously, every XVSM request can be interpreted as an individual event, as in publish/subscribe or event-based systems. Therefore, this category refers to the usage of dataflow architectural styles and implicit invocation architectural styles (see sections 2.3.3.1 and 2.3.3.4).

Aspects are used intensively in this category and can be further divided into three types of responsibilities:

- aspects contain logic that is restricted to the aspect only, like filter functionalities.
- aspects execute logic with access to other containers, like in the case of aggregating events or logging operations. If aspects access other containers, they are handled as if they were ordinary application components from the computation category.
- aspects require access to third-party resources, like to a database or to a Web Service to verify operational parameters.

As an example, container C2 in the organization category in Figure 28 represents a Virtual Container. Incoming operations are not retrieved from or written into the container, but they are forwarded to a database. The application component thinks, it deals with container C2. This allows e.g., to transparently switch from a database connection to a Web Service call, even without changing the parameters of the selector.

4.2.4.4. Distribution Category

The distributional category deals with transparency issues known from distributed systems, like location-, migration-, relocation-, and replication-transparency [216]. From the application point of view it is not known whether the container is embedded, hosted on a remote server, or replicated among several nodes. By means of aspects the container can be placed, located and replicated in a P2P environment. In such case, aspects installed at every replica would define the replication technique and the consistency strategy between the replicas. They reuse existing solutions to these issues, like in group communication protocols.

Which replication strategy is the most suitable for a container depends on the use case. For instance, in scenarios where security constraints specify that every operation has to be carried out at a specific location, publish/subscribe systems would be sufficient to propagate changes. In scenarios, where every replicated container allows write operations, group communication protocols are needed for keeping the participating containers consistent. An example of how to realize replicated containers in a P2P network transparent to the application is explained in section 5.1.

4.2.4.5. Communication Category

In this category XVSM allows to specify how information is exchanged between the participating containers. The communication category describes how data from one container is transmitted to the other and provides mechanisms to lookup the physical network address of container URIs. In the category there may be lower level protocols

like udp, tcp, and higher level ones like P2P protocols. It also allows the deployment of customer specific transport protocols.

4.3. *Mapping Architectural Styles*

This section describes how the various architectural styles specified in section 2.3.3 can be mapped to the concepts of the XVSM architecture.

4.3.1. Data-centric Architectural Styles

In the data-centric architectural style components communicate with each other via a central data store. In XVSM application components communicate with each other by means of writing entries (i.e. data) into containers (i.e. central data store). Therefore, data-centric architectural styles and the SBC architectural style overlap each other.

Blackboard

The Linda coordination model is an example for a blackboard based architectural style. XVSM uses the primitives of tuple spaces and additionally extends the Linda coordination model by flexible coordination policies. Therefore, XVSM inherently supports the blackboard based architectural style.

Repository

A shared repository provides its clients access to shared data by means of an API or query language. Databases, usually supporting the SQL standard, are the typical representation of this data-centered architectural style. Analogous to the *insert* operation of a database, the *write* primitive in XVSM adds new data to the repository (i.e. container). The database methods *delete*, and *select* use queries comparable to the primitives *delete*, *take*, and *read* in XVSM that use selectors specifying the parameters of the query executed in the coordinator. An *update* database operation is not provided by XVSM, but can be mapped onto the XVSM operations *take* and *write*. There are two ways (Figure 29), how a repository architectural style can be realized with XVSM concepts. The difference between the two ways is how much database technology is encapsulated in XVSM concepts.

The first possibility (Figure 29A) uses a virtual container and a pre-aspect that has access to a database. The database is in that case a third-party resource whereas the aspect is responsible for establishing the connection to the database, executing queries, and returning the results of the queries. The pre-aspect receives the XVSM requests and

maps the request according to the type of the operation and the context parameters to a database query. Then, the transformed request is sent to the database.

The second possibility realizes a repository architectural style by means of a coordinator (Figure 29B). In this case there are two more ways to be considered. Either the coordinator itself implements a database or as the pre-aspect in Figure 29A transforms operations to queries and forwards them to the database. The difference between these two options is the amount of implementation effort that has to be invested. It is easier to establish a connection to an external database than implementing one from the scratch.

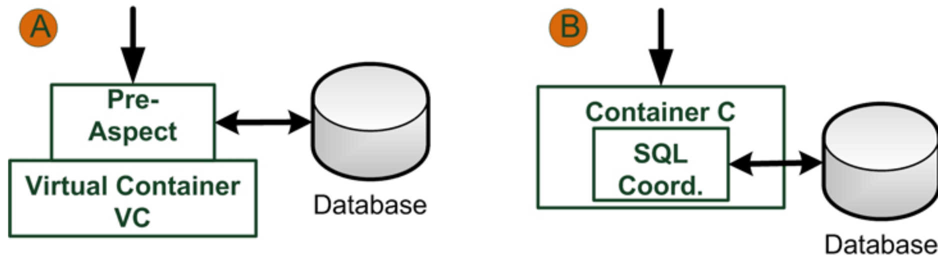

Figure 29: Repository architectural style realized with XVSM concepts

The difference between Figure 29A and Figure 29B is the way how *write, read* and *take* operations are treated. In case A, if the result set of a *read* operation is empty or the *take* operation failed the pre-Aspect would return the empty result set or the appropriate error. However, in case B the coordinator will indicate that the operation has to be blocked. In case A, if a *write* operation is performed, the pre-aspect adds the data to the database. In case B, the coordinator has the ability to execute a query – that asks whether the data to be written already exists - before inserting the new data. The query makes sure that the data to be written does not exist. If it exists the coordinator requests the XVSM runtime to block the write operation.

Replicated Repository

In the replicated repository architectural style several processes providing the same service interact to improve the accessibility of data and scalability of services of a system and to present the illusion of a centralized service. There are two ways how a replicated architectural style can be realized with XVSM concepts.

In the first approach, each data store is represented by one virtual container and accessed as described in the previous section and shown in Figure 30 or Figure 29A. Whenever an application component executes an operation, it is transformed to a query and executed on the database. Using this approach the database is in charge of keeping the replicas consistent.

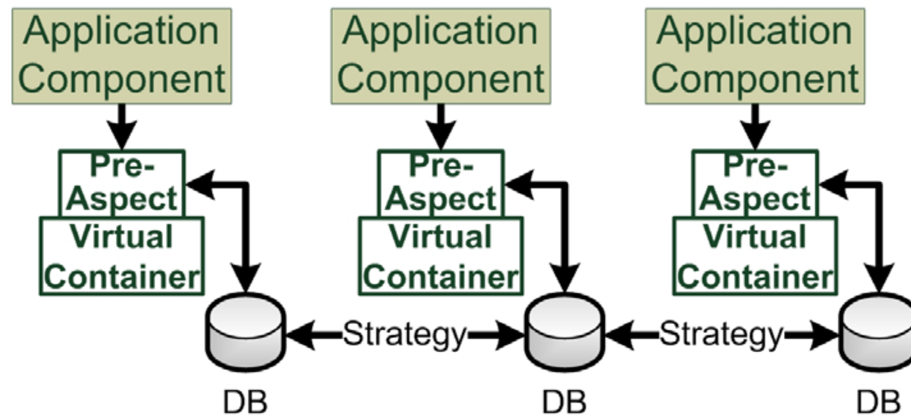

Figure 30: Replicated repository architectural style using database specific strategies for consistency management

The second approach is shown in Figure 31 and requires the usage of two pre-aspects. In contrast to the previous concept, pre-Aspect 1 is responsible for keeping the replicas consistent. This is achievable by e.g., implementing the same strategy as the database (see Figure 30). In a simple strategy, pre-Aspect 1 lets through read operations to be executed locally. Manipulating operations like write, take, and destroy are first copied and forwarded to the other pre-aspects. Once replication of the operation has been finished successfully the operation is executed locally.

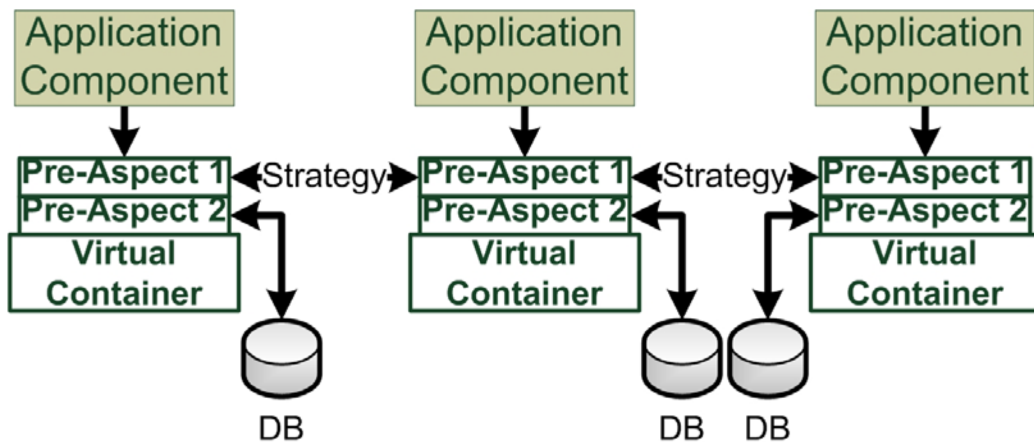

Figure 31: Replicated repository architectural style using pre-aspects for consistency

The difference between the two approaches relates to complexity and flexibility. In both approaches the strategy of consistency management is abstracted in extension profiles implemented by a software system developer. The first approach facilitates low complexity and provides low flexibility. Since replication and consistency issues are built in features of the database there is minimal implementation effort (i.e. low complexity) required from the software system developer. Flexibility is restricted because the given strategy is defined by the database. In the second approach replication and consistency issues have to be implemented by the software system developer

resulting in high complexity. Therefore, the concept facilitates flexibility since the implemented strategy can be fully adjusted to business requirements.

4.3.2. Dataflow Architectural Styles

Dataflow architectural styles are concerned with the movement of data between independent processing elements. For the mapping of dataflow architectural styles, aspects are used again.

Batch Sequential

In the batch sequential architectural style processing steps are independent application components reading data from a source and writing data to a sink. Data is transmitted as a whole between application components whereas each step runs to completion before the next step starts. Figure 32 shows the mapping of that style to XVSM concepts. Application components write data (i.e. entry) into a container hosting a FIFO Coordinator. In fact, any coordinator that does not require any parameters when writing or reading an entry is suitable because the container is bounded to a single entry. It is bounded to make sure each step finishes before the next step may commence. This means that as long as an entry has not been *taken* by e.g., Application Component 2, Application Component 1 is not able to *write* an entry.

Figure 32: Batch sequential architectural style realized with XVSM concepts

Pipe and Filter

In the pipe and filter architectural style processes read the relevant input stream and write the required output stream. Filters incrementally transform data from input stream to output stream. Filters can enrich data or refine data. Mapping that style to XVSM concepts requires a container hosting a FIFO Coordinator (Figure 33).

Figure 33: Pipe and filter architectural style realized with XVSM concepts

A container between two application components (i.e. filters) represents the stream. The responsibility of the FIFO Coordinator is to queue incoming data according to the order they were written, and to return data when application components read their input stream. However, it is important that the semantics of the FIFO Coordinator correctly

portray the temporal relationship between written entries, as it is fundamental to data streams.

Message-Queuing

In the message-queuing architectural style the sender application component and the receiver application component are decoupled. They do not need to know each other's location nor identity. Decoupling is achieved by placing messages into a common queue. Mapping of that architectural style to XVSM concepts is achieved by placing a container hosting a FIFO Coordinator between the application components (Figure 34). The semantics of that coordinator is the same as of a queue. Placing a message into a queue is mapped to writing an entry into the container with the FIFO coordinator. Retrieving a message from a queue is mapped to a *take* operation on that container. The FIFO semantics of the coordinator makes sure that entries are received in the same order as they were written.

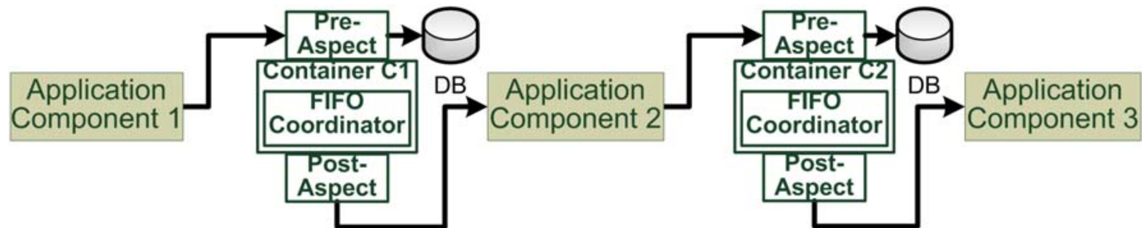

Figure 34: Message-Queuing architectural style realized with XVSM concepts

Figure 34 shows an approach for reliable messaging between two application components. The pre-aspect intercepts write operations and places the written entry into the database before passing it to the coordinator. Post-aspects intercept *take* operations and remove the entry from the database.

4.3.3. Explicit Invocation Architectural Styles

The explicit invocation architectural style synchronously connects two application components. Since the XVSM architecture operates asynchronously, synchronous communication needs to be simulated. A client application component requests services from the server application component. In XVSM there is no direct communication allowed between application components. Communication takes place by asynchronously writing and reading data from containers.

There are two ways how synchronous communication between two application components can be realized. The first approach (Figure 35) uses answer containers while the second approach (Figure 36) combines the power of two coordinators.

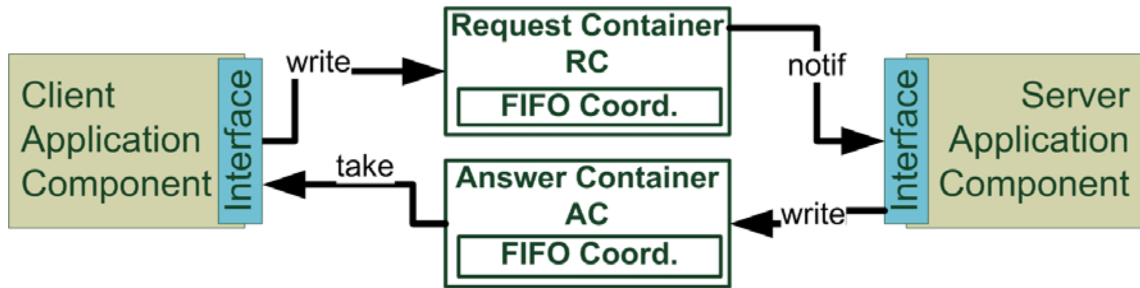

Figure 35: The client/server architectural style realized with two containers

In the first approach, the requesting client writes its request into a request (RC) container using a FIFO coordinator. The request also contains information about the answer container where the client expects the server's result. The answer container also hosts a FIFO Coordinator, thus it is made sure that results are provided in the same order as requests were written. The server has two options to get the request from the RC container. Either it performs blocking *take* operations or it specifies a callback method to receive consuming notifications. Like in the figure, a consuming notification notifies the server about changes, provides the data that has been written into the container, and removes that data from the container. When the server is finished with its calculations it writes the result into the answer container (AC). Once the client has written the request into the RC container, it executes a blocking *take* operation on the answer container. When the server's result is written into the answer container, the blocking *take* operation is released and the result is taken by the client application component.

Figure 36: The client/server architectural style with two coordinators

The second approach combines two coordinators (FIFO and Key) on a single container. In the previous approach the URI of the answer container is used by the client to identify the location of the result. In this approach the client specifies an identifier at each *write* operation that is used by the server as parameter for the Key Coordinator when it writes the result into the container. The client application component writes its request into the container using the FIFO Coordinator. The identity of the entry is embedded in its Key. Again, the server application component either performs blocking *take* operations or it specifies a callback method to receive consuming notifications. When the server is finished with its calculations it writes the result into the same container using the identifier of the request as a parameter for the Key Coordinator. The client retrieves the result by invoking a *take* operation with the identifier as parameter in the selector of the Key Coordinator.

Explicit calls require that the clients are aware of an unavailable server. Due to decoupling of components it has to be made sure that the client gets notified in case the server component does not exist. This is in both approaches achieved by the timeout the software developer specifies when invoking an XVSM operation. If the timeout expires the application component receives an exception. However, this does not really indicate that the server is down. It may also refer to a slow server taking too long to calculate and write the result to the request. In that case the client may receive an exception although the result is in the container.

The difference between the two approaches is the number of containers needed and how exceptions are treated. The first approach allows the dynamic definition of answer container locations with respect to client specific requirements implying the possibility of several answer containers at the same time. Requests and responses are decoupled minimizing the risk of communication errors due to network failures. Even if one container fails (due to the crashed node that hosts the container) the other container is still accessible. With respect to error handling, the FIFO coordinator in the first approach increases the complexity of the client. After a blocking *take* operation is released due to a timeout exception, the client application component might drop its interest in the result. However, if the next time the *take* operation is invoked, the client may find the result of the previous request. Therefore, the client application component has to implement logic that is capable of filtering invalid results. In case of the Key Coordinator the client application component is able to exactly specify the required results. However, the client also has to make sure that each request has its own unique Key and that Keys are stored safely for retrieving results after recovery.

4.3.4. Implicit Invocation Architectural Styles

The implicit invocation architectural style is characterized by calls that are invoked indirectly and implicitly as a response to an event. In XVSM decoupling is supported by a container using notifications as described in section 4.2.3.2.

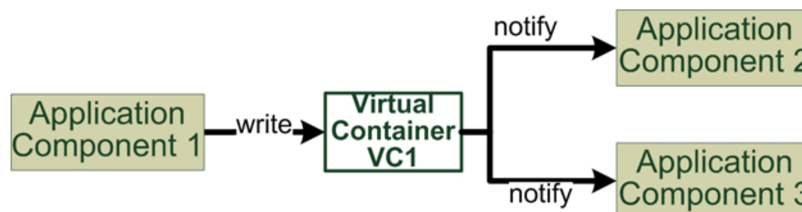

Figure 37: Implicit invocation architectural style

Figure 37 shows a publisher (Application Component 1) and several subscribers (Application Components 1 and 2). The publisher *writes* events/messages into the container VC1. Whenever a publisher writes into the container subscribers are notified. The container is virtual because it does not need to store any events. Any incoming

event is copied and written into the notification container of each of the subscribers, as described in section 4.2.3.2. This concept works well for a small number of subscribers if the virtual container is hosted on a single server.

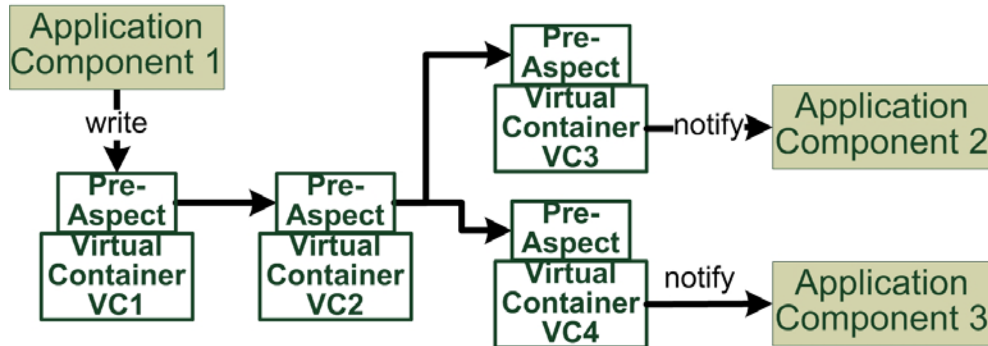

Figure 38: Implicit invocation architectural style in larger networks

In case a lot of subscribers have to be notified the virtual container has to be extended by routing logic as shown in Figure 38. In that figure, it is assumed that each of the shown containers is hosted on different spaces on different nodes. Each of the virtual containers has a pre-aspect knowing where to route incoming events. Strategies for routing messages to subscribers efficiently is described in [68]. A message has reached a subscriber if that particular virtual container has a notification aspect with matching notification subscriptions on the entry (i.e. published message) written.

Finally, it is noted that the semantics of a notify operation differ with the used operations. As described in [127] and [186], this results in different so called notification flavors, resulting in different flavors of publish/subscribe approaches.

4.4. *Discussion*

In this chapter the Space-Based Computing architectural style and the concepts of its reference architecture called eXtensible Virtual Shared Memory (XVSM) were described. It detailed the mapping of SBC architectural components like coordination policies or extension profiles onto XVSM architectural concepts like coordinators and aspects. Coordinators serve to implement coordination requirements, and aspects to implement extension profiles. It was shown that XVSM's simple API and both concepts are sufficient to abstract various architectural styles. In the following it is discussed how SBC relates to other coordination frameworks and why it helps to cope with agile requirements affecting software architectures.

4.4.1. Interaction

The Linda coordination model exhibits the problem that access to local tuples is tied to the built-in associative mechanisms of tuple spaces. This implies that any non-directly supported coordination policy, like automatically reading several tuples, has to be charged upon the coordinating processes. This means that processes have to be made aware of the coordination laws increasing the complexity of the application design and so breaking the separation between coordination and business issues.

Besides XVSM the LuCe coordination framework, described in section 2.2.3, offers the possibility to enrich the semantics of coordination operations. XVSM achieves this property by means of changeable coordinators. LuCe relies on the usage of so called reactions. Such reactions are hidden from the application and triggered transparently to the application whenever an entry is written or read. Reactions are also changeable and capable of simulating any kind of coordination policies.

However, LuCe just maps a single logical operation onto one or more system operations. This means that one operation in the application is mapped onto several Linda operations in the system. Therefore, the complexity of coordination policies has been just moved from the application (and consequently from the application developer) to the coordination middleware, thus to the software developer of that platform. In contrast, XVSM also moves the complexity of coordination from the application to the coordination middleware, but allows the usage of language specific primitives (i.e. the semantics of a FIFO coordination can be mapped on java.list) which shift complexity further away from the software developer into the compiler of that language.

For example if LuCe had to support coordination models with ordering requirements, every tuple in the space had to be additionally wrapped into a tuple managed by reactions. This extra tuple stores meta-information, like the position in the queue, of each written tuple. Consequently, every incoming operation has to be adapted according to the new structure of the tuples, which decreases performance. In the MozartSpaces implementation of XVSM Java specific functions/libraries are used to organize entries in a queue resulting in a single and efficient operation.

Based on the fact that reactions in LuCe are implemented by means of Linda primitives, they cannot access other resources but the tuple space. Aspects in XVSM are written in higher-level languages and allow therefore the integration of other technologies, like web services or databases, into the coordination process. Furthermore, reactions cannot execute blocking operations. The limitation may arise due to the missing separation between reactions responsible for coordination and reactions responsible for e.g., tuple aggregation. Reactions must be non-blocking since strategies for synchronizations of reactions had to be implemented which would significantly decrease the performance of the platform.

4.4.2. Agility

In sections 2.1.4, 2.3.2, and 2.4.3 several methods were described how to deal with complexity and agility issues in software systems for clean software architectures. The XVSM reference architecture uses concepts which follow these guidelines. In the following it is explained how the guidelines have been realized in XVSM in order to demonstrate the architecture's capability of acting as a solid platform for creating complex software systems. The discussed aspects relate to: simplicity, abstraction, decoupling, high cohesion, separation of concern, information hiding, and design for change.

Abstraction and Simplicity: XVSM provides a basic application API consisting of *read*, *take*, *write*, *destroy*, and *notify* operations. Although the number of methods is higher in comparison to e.g., message passing systems (*send*, *receive*), the number of concepts (container, coordinator, selector, and aspect) and primitives (*read*, *take*, *destroy*, *write*) remain unchanged in every architectural style XVSM is able to represent.

Decoupling: The XVSM architecture facilitates decoupling of application components in several ways. In XVSM application components may run in different computational environments (space decoupling), interacting or coordinating application components do not need to know each other (reference decoupling) or be running at different times (time decoupling). Additionally, XVSM decouples application components without the need for explicit synchronization in the process of coordination (coordination decoupling).

High cohesion: Architectures making use of the high cohesion principle allows architectural elements to be interpreted as black boxes which can be manipulated and adapted independently of other elements. XVSM facilitates high cohesion in coordinators uniting all aspects of coordination at one place. This allows software engineers to analyze trade-offs and tune the way of coordinating application components according to scenario specific circumstances.

Separation of concern is supported in the sense of the five categories of decoupling. The architectural concepts of XVSM are clustered and categorized in a way in which coordinators and aspects have their specific responsibilities to deal with issues of distribution (like location, migration, relocation, and replication transparency) and agility of complex software systems.

Information hiding provides only those parts of information which are really needed by other elements whereas remaining parts remain hidden. XVSM supports application components in their ways of coordination. They only need to know their way of coordination but not the specific implementation.

The architecture of XVSM facilitates **design for change** by enabling to switch coordinators, or to extend coordination capabilities between two application

components by adding additional coordinators. Furthermore, XVSM provides the capability of deploying scenario specific plug-ins, known as extension profiles.

Chapter 5

5. Prototypic Realization of the Application Scenarios

This chapter describes how the three application scenarios have been designed by using XVSM concepts and implemented as a prototype. The first scenario (RealSafe) demonstrates SBC's coordination capabilities under strict timing restrictions and its efficiency as data-centric architectural style, the second scenario (SWIS) shows the agility capabilities of the architecture due to changing requirements, while the third scenario (SAW) reports robust coordination needs for dynamic application components in dataflow architectural styles.

5.1. *RealSafe*

The aim of the Real-time Safety-related Traffic Telematics (REALSAFE) project is to tackle the challenges for scalable and dependable vehicle-to-infrastructure (V2I) telematic systems in the Intelligent Transportation System (ITS) domain.

5.1.1. Requirements concerning the Architecture

With respect to the domain scenario described in section 3.3.1, architectural requirements refer to

- the ability to collect contextual data taking into account geographic information
- appropriate generation and transportation of comprehensible, reliable and timely content to users
- the complexity regarding information retrieval from a various number of involved data sources (e.g., sensors, control centre) under timing restrictions of mobile users
- properties like scalability and self-organization the architecture has to satisfy to increase fault-tolerance and availability.

5.1.2. Proposed Architecture

As shown in Figure 14 the road network is separated into several road segments where each RSU maintains at least one road segment. Traffic information is bound to a geographic area and refers to a road segment. An RSU is responsible for transmitting and retrieving information regarding the road segments to and from vehicles passing by.

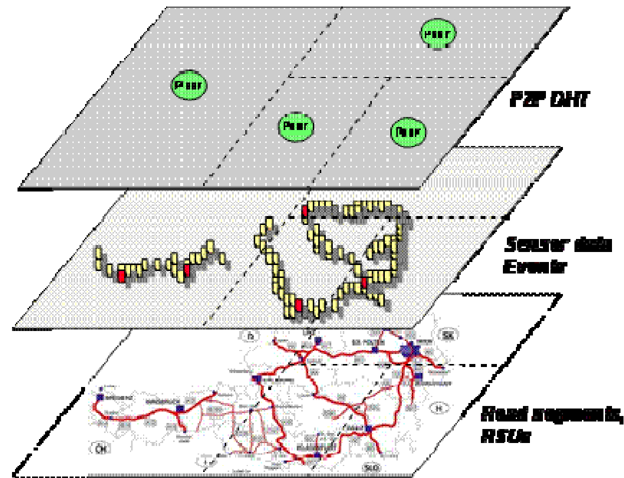

Figure 39: Relation between road network – road segments - RSU

The effectiveness of decision support systems [215] depends on the ability to collect contextual data from many different sources (e.g., sensors, cameras, or personnel). Structured P2P networks, like Chord [213], Pastry [191], or CAN [189], use Distributed Hash Tables (DHTs) and have been extensively studied and successfully deployed to create scalable and fault-tolerant applications. A few P2P techniques such as the intentional naming system INS/Twine [14] or P-Grid [1] can maintain a structure with a hierarchical DNS-like addressing. However, the costs for such solutions are high, since keeping a P2P entry for each data item (or message) causes a large traffic overhead due to internal reorganization of references, especially when the data elements are mutable and short lived. However, when structured content, such as context data with temporal and geo-location attributes needs to be efficiently stored, queried and acted-upon, a storage network design that supports mainly files renders unsuitable, since the information structure gets lost.

For this project it has been decided to integrate Distributed Hash Tables (DHT), in particular an implementation called Pastry [191], and the XVSM architecture. The architectural approach for the scenario combines XVSM containers with an overlay network consisting of RSUs based on DHT concepts (Figure 39). Containers provide coordination support, query expressiveness, and mechanisms for manipulating complex, dynamic data objects (i.e. traffic information) while Pastry makes such containers uniquely addressable in a fault-tolerant and scalable manner.

DHTs are suitable due to their scalability characteristics and the efficient demonstration of self-organizing properties in case network peers (RSUs) are added or removed. DHTs achieve a fast and scalable behavior because they support key-value pairs only. This means that a value stored in the network can only be retrieved if the key for that value is known. Therefore, structured information cannot be efficiently handled by means of DHTs only. The combined architecture enables applications that operate on distributed storage of structured data requiring distributed coordination.

5.1.3. Limitations of Related Technologies

In the following, the section illustrates an alternative approach to the proposed architecture. It describes the advantages and limitations of DHTs regarding retrieval of data. Furthermore, it is discussed why current publish/subscribe systems regarding communication between mobile users and databases functioning as storage and aggregation components of traffic information instead of XVSM containers and aspects are not fully suitable for the domain.

5.1.3.1. Centralized Architecture

As an alternative to the proposed distributed architecture, a centralized architecture approach based on existing mobile telecommunication standards (e.g., UMTS, HSDPA, or LTE [91]) and infrastructure is shown in Figure 40.

Although the architecture allows a distributed access to the network infrastructure, the architecture itself does not provide a fully distributed solution. It has a central architectural component, called gateway, which connects the network infrastructure via the Internet. As the project would like to offer multimedia content and full access to Internet services in the future, that central component is interpreted as a bottleneck and a single-point of failure.

Another limitation refers to the property how such communication standards work. Typically the uplink speed between users and network infrastructure is limited restricting the amount of data collected by sensors for upload. Furthermore, the available bandwidth for downloading information has to be shared between the users belonging to the same network cell. Typically, network cells cover a large region, and therefore they also have to manage a large number of users. Consequently, this results in less bandwidth available for each user to receive traffic information.

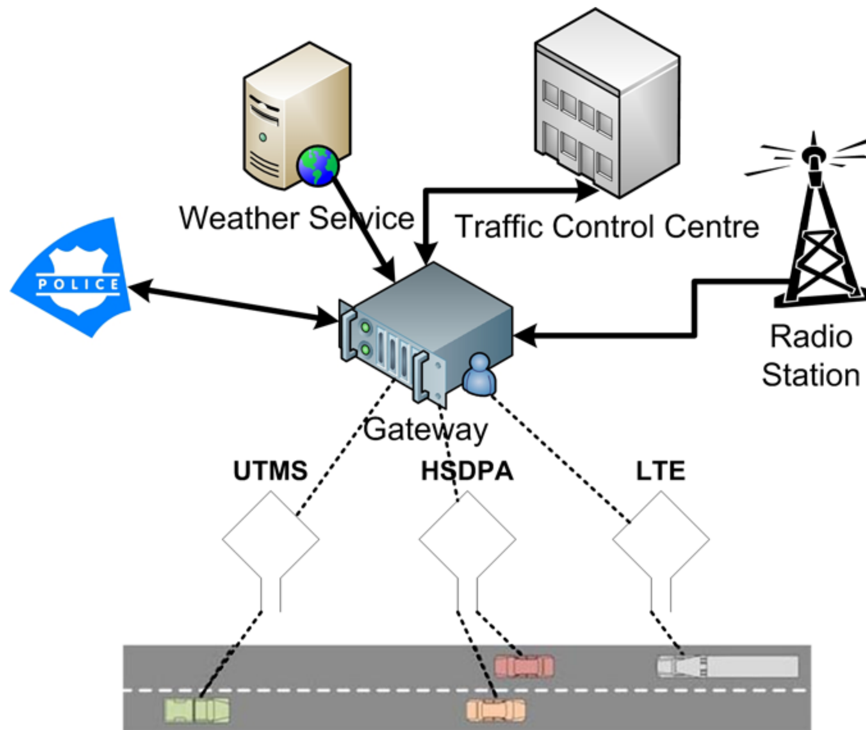

Figure 40: A centralized architecture approach

5.1.3.2. Distributed Hash Tables

Distributed hash table (DHT) is widely used in structured peer-to-peer networks. Examples are Chord [213], Pastry [191], or CAN [189]. Data in DHT can be addressed and stored without a central server. The basic provided functions of a DHT are to store a data object and to retrieve it efficiently from any peer in the network. For this purpose, a very large identifier space (e.g. all binary combinations of 128 bits) is defined onto which data objects and node identifiers are mapped using a hash function. The design options to create the key for a data object vary from hashing the (string) name of the object, its location (URI), or the content itself. Every node in the DHT takes charge of the routing issue in a certain area and stores data according the data keys that map to this area. The data record (K, V) has an identifier P = hash(K), where P designates the position of data in the network.

DHTs take care of

- efficient distribution of the data in the key space for load sharing purposes
- replication for fault-tolerance reasons

Therefore, a DHT offers basic self organization functions, such as adapting the topology when a node arrives or leaves, and a increased reliability since it has mechanisms in place to replicate DHT entries. Unlike the unstructured peer-to-peer system, this feature is more suitable to build high-level services on top of overlay networks because the

routing cost are more predictable. Moreover, DHT lookups can be resolved in O(log N) efforts where N is the number of nodes in the overlay network. However, with DHTs it is only possible to lookup exact values, but it is not possible to include wildcards or any other query expression for retrieving values.

5.1.3.3. Publish/Subscribe Systems

Publish/subscribe systems are another technology to be considered for the ITS domain. After providing an introduction to publish/subscribe systems, their limitations with respect to the scenario are described.

The publish/subscribe paradigm defines two types of clients: publishers, which are generating events/messages, and subscribers, which receive notifications of events/messages they have previously subscribed for. This type of messaging paradigm allows decoupling of publishers and subscribers in time and space. Furthermore, it enables asynchronous and anonymous communication (i.e. reference decoupling) between publishers and subscribers [68].

Publish/subscribe systems are distinguished into two types: topic-based and content-based. In a topic-based publish/subscribe system subscribers can only register their interest on specific topics (mostly predefined) under which publishers are dispatching their generated events/messages. Whereas, in content-based publish/subscribe systems subscribers are not constrained to specific topics, they can define more precisely in what kind of events/messages they are interested in. On the one hand, this provides more flexibility to the publish/subscribe system but on the other hand the publish/subscribe system has to match events/messages to the subscriptions [237]. Hybrid publish/subscribe systems like Hermes [184], SIENA [135] or Rebecca [37, 70] provide both: topic- and content-based subscription. With respect to the scenario a subscription

The publish/subscribe system architecture can be classified into: client-server and peer-to-peer. In a client-server architecture publishers and subscribers are both clients which are connected to a network of servers. Generally, the servers temporarily store the events/messages generated by publisher-clients and forwarded them to the subscriber-clients. If a subscriber-client is not directly connected to a server, where a publisher has dispatched an event/message, the server has to forward the event/message to another server. The forwarding between the servers is repeated until the event/message reaches the server, which is able to deliver the event/message directly to the subscriber-client. Gryphon [240], for example, is a context-based publish/subscribe system with a client-server architecture, which uses so called brokers as servers. The brokers are responsible to determine to which subset of brokers the event/message should be sent. In a peer-to-peer architecture all nodes are equal, meaning that each node can act as a publisher, subscriber or event/message-forwarder to another node. SIENA is a mixed form of

client-server and peer-to-peer architecture, because publishers and subscribers are clients but servers are working together in a peer-to-peer topology.

Since network infrastructures are not immune to network failures or crashed network nodes, publish/subscribe systems (both client-server and peer-to-peer architecture) have to be reliable and fault-tolerant. Reliable publish/subscribe systems guarantee that published events/messages are delivered to all subscribers. Durable (fault-tolerant) publish/subscribe system are able to cope with unreachable subscribers and servers. Some publish/subscribe systems like SIENA offer a best-effort delivery strategy. This means that the system will periodically retry to deliver the message until the message was delivered successfully, a timeout expired or the maximum retry-count was reached.

The strong decoupling and the asynchronous and anonymous messaging provided by the publish/subscribe paradigm makes it very attractive to be used in mobile environments [103]. Client applications reside on a host that is moving and therefore accessing the network (composed of so called event brokers) from various locations [21]. The event brokers are responsible to guarantee the reliability and durability of the publish/subscribe system as described before. Furthermore, a protocol must exist which enables the update of a client's subscription as it is moving from one broker to another. During the client's movement undeliverable events/messages have to be stored by the system and delivered as soon as the client reconnects to the system. When the client reconnects at another broker, all stored events/messages have to be forwarded to the broker where the client is actually connected. The authors of [227] propose a two-phase handover (2PH) protocol, which reduces network traffic and the latency when a client reconnects to the system. One of the first publish/subscribe systems that supports the reconnection of mobile clients is JEDI [240]. Later, existing publish/subscribe systems like SIENA and Rebecca have been extended to support mobile clients [39, 54, 167].

Mapping the concepts of a publish/subscribe system capable of functioning in a P2P environment with mobile clients to the given scenario, means that e.g., the Traffic Control Centre is a publisher, and RSUs are event brokers. Events/messages refer to information about e.g., car accidents to be delivered to subscribers. Depending on the type of subscription, there are two possibilities for a subscriber. In case of a content-based subscription, e.g., vehicle drivers are subscribers. In case of topic-based subscription, an RSU is a subscriber, whereas the topic of the subscription is the region the RSU is responsible for.

With respect to vehicle drivers as subscribers the following has to be considered. Although publish/subscribe systems, ready for mobile environments, are capable of coping with offline subscribers, those subscribers are not only mobile but also most of the time offline. In fact, being disconnected from the system is the rule rather than the exception. This implies that the number of events the publish/subscribe system has to cope with for each subscriber is increasing with the time the subscriber is offline. This has the indication that at some point in time the system gets into a kind of back-pressure

and has to drop messages which is not allowed due to the safety-critical characteristics of the domain. Furthermore, every time the subscriber connects to a new broker, its subscription has to be updated, and already stored events have to be retrieved from the broker where the subscriber was previously registered. Since subscribers in this domain move from one RSU to the next, these processing steps have to be executed every time the subscriber reconnects to the system. This increases the amount of traffic transmitted between the RSUs significantly, and minimizes the time available for successfully pushing events/messages to the subscriber who is moving fastly and therefore has only a few seconds left for data transmission.

In case an RSU is a subscriber for regional specific information, the topic of the message specifies the target road segment. Since an RSU knows its road segment it subscribes for the appropriate topic. Every message that is received via that topic is transmitted to passing by vehicles. However, the big disadvantage of this approach is that every time an RSU is added to or removed (manually or due to a failure) from the system, remaining RSUs have to manage responsibilities regarding involved road segments automatically. If an RSU is removed its road segment has to be taken over by the RSU next to it. If an RSU is added then subscriptions and topics have to be adapted in the sense that each topic is used by one RSU only. In case multiple RSU subscribe for the same topic an event is transmitted multiple times which violates safety-critical characteristics of the domain.

5.1.3.4. Databases

For storage of events databases would be a possible technology of choice. In fact, by means of triggers instead of aspects aggregation of events is possible as well. Therefore, the question is whether databases could have been an alternative, instead of containers in the described scenario?

The problem is that established enterprise database products are heavy-weight components, and as such the peers in the network do not have the capacity to run them. In this sense, an alternative would be embedded, light-weight databases, like Oracle Berkeley DB Java Edition[11], Apache Derby[12], hsqldb[13], Axion[14], H2[15], or db4o[16].

---

[11] http://www.oracle.com/database/berkeley-db/index.html

[12] http://db.apache.org/derby/

[13] http://hsqldb.org/

[14] http://axion.tigris.org/

[15] http://www.h2database.com/

[16] http://www.db4o.com/

However, a reason to prefer containers rather than database technologies in this context is the fact that the types of events in the ITS domain is not known beforehand and as a consequence an appropriate data model is difficult to establish. Databases need a static data model of the entries they have to store while a container allows the usage of several different coordinators at the same time enabling ’dynamic’ data models which can be plugged in whenever needed. Finally, databases aim to store and retrieve long living data, while the scenario focuses on short lived data.

db4o is an object-oriented database. Accessing an entry is performed via query-by-example, comparable with template matching in tuple spaces, and therefore more suitable for enabling dynamic data models. However, the structure of traffic information is plain xml rather than Java classes.

5.1.4. Description of the proposed Architecture

This section describes the architecture and functionality of the integrated approach in detail.

5.1.4.1. Integration of DHT and XVSM

The reason why Pastry has been chosen for the integration with XVSM is that in contrast to other DHT implementations it provides events about its internal processes to the application using it [55]. The integration of Pastry and XVSM is done in two steps [85]. The first one includes the mechanisms of Pastry in the communication category of XVSM as a transportation profile. The second step makes use of its self-organizing capabilities and of its published internal events in the distribution category of XVSM.

Communication Category

The given addressing scheme for XVSM (section 4.2.2.4) allows applications to directly access a container in case the URI could have been resolved. In the proposed architecture it is assumed that the key-value pair entry in the DHT overlay network consists of the name of the container as key (“containerName”), and of the reference to the container as the key’s value (containerReferenceURI). However, a fixed physical identifier (i.e. IP address) binding leads to known problems in case the IP address changes (due to mobility of the node), the node fails or if containers are moved to other nodes. Therefore, the stored reference (i.e. the value to the key) has the format tcpjava://localhost:1234/ID, where “localhost” is just a place holder for an IP address, and ID uniquely identifies the container.

In contrast to an ordinary DHT lookup where the value to a key is returned to the application requesting the lookup, the Pastry based XVSM transportation profile makes

use of Pastry's internal events. Whenever an application requests the value for "containerName" using the DHT overlay, the Pastry transportation profile of the destination node completes the containerReferenceURI by replacing "localhost" with the actual IP address of the destination node, before returning the result to the requesting application.

Concerning the geographic distribution of the RSU nodes and the geographic relevance of messages, the ordinary Pastry DHT hash function does not support this requirement. In Pastry, data set placement and routing is done by defining and slicing an 128 bit ID space. A hash function is used to map nodes and entries to these IDs. The default implementation is to utilize the well known SHA1 hash function to encode the keys of an entry. We focus on spatial-temporal content that, in addition to its similarity to messages or events, is semantically related to a geographical pair of (WGS84) coordinates. In some services it is required that a piece of content shall be routed to a node responsible for those coordinates.

To realize this kind of functionality, a hash function that takes geographical location into account is superior to the random behavior of standard SHA1 hash function. In order to build the location-aware-key, an identifier called Cname (i.e. containerName) is added to the coordinate pair, because the coordinates alone cannot qualify collocated objects. This concept leads to a hash function signature similarly to [99]: H(x; y;Cname), where x refers to longitude and y to latitude geographic references, and Cname is the identifier already mentioned.

Distribution Category

The distribution category of XVSM deals with transparency issues related to distributed systems (section 4.2.4.4). For a scalable and fault-tolerant system it is important to maintain several replicas of the data to be accessed. However, in consideration of the proposed architecture, this implies that both DHT key-value pairs and the containers have to be replicated. In the following it will be distinguished between replicas of DHT entries (DHT replica) and replicas of the containers (container replica).

With respect to maintaining DHT replica the proposed architecture relies on Pastry's built-in mechanisms. Pastry already provides strategies for creating new DHT entries and migrating existing ones to other overlay network nodes in case the structure of the network changes. Keeping DHT replicas consistent is not required since the value of a key is not updated. Whenever a replica is created or removed Pastry provides internal events which are used to manage container replica. The *replicaSet* operation offers information about where the set of nodes in the network hosting replicas for a given key.

In case nodes fail, the lost DHT entries are replicated in order to keep a predefined number "r" of copies for each entry, thus Pastry automatically creates new DHT entry replica. In addition, whenever an internal event indicates that Pastry has added a new

value to a key to the current local node then a corresponding container with the appropriate ID is created and synchronized with the other containers. Whenever an internal event indicates that a key has been deleted, the appropriate container is removed as well.

5.1.4.2. Container Replication

Containers are components in which entries change due to operations like *write*, *take*, or *destroy*. Therefore, the container replicas have to be updated, although DHT values remain unchanged. A single master replication scheme cannot be used because the design of the interactions related to the operations has to consider the replication mechanisms specific to the DHT implementation. The reason is that it is never known which destination node is returned by the lookup operation of Pastry. As an example the delegation pattern is used (Figure 41) as an appropriate replication strategy which can be replaced by other replication strategies by changing the aspects.

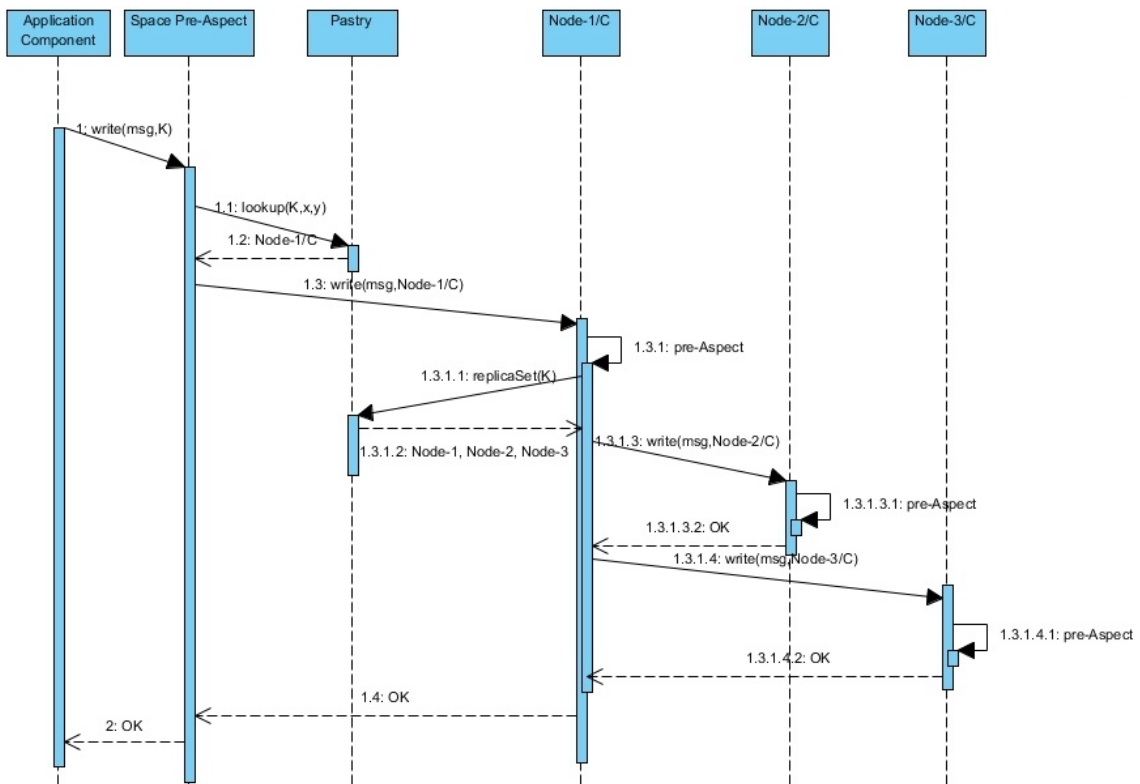

Figure 41: Handling container replicas [123]

The sequence diagram shown in Figure 41 is a representative for all XVSM operations manipulating the content of a container (i.e. *write*, *take*, and *destroy*). It shows the processing steps in case a write operation is invoked by an application component (i.e. TCC) that wants to insert the message “msg” into the container named “K”. Since

containers are distributed in the overlay network, their specific location is not known for the application. The application can only specify the name of the container. For application developers the API is a compact, elegant abstraction that needs only the application specific container name for addressing it network-wide. The write operation of the application component is intercepted by a space pre-aspect (section 4.2.2.1). The aspect performs a DHT lookup operation using "K" as the key and the geographic information extracted the message (step 1.1). The result of the operation is a URI (i.e. Node-1/C) of a container (step 1.2). The next operation of that pre-aspect proceeds with a direct container *write* operation using the retrieved URI (step 1.3). Additionally, the aspect specifies the context of the operation by adding the current container's URI. On the invoked container a replication pre-aspect intercepts the *write* operation. That aspect is responsible for informing the other container replicas about the new operation. It requests an update on all existing replicas regarding the key "K" by invoking the *replicaSet* operation. In the scenario the key is replicated three times, therefore three containers exist in the space. The operation returns the location of the replicas in the network (step 1.3.1.2). The aspect uses this information and executes two more write operations (steps 1.3.1.3 and 1.3.1.4). However, it also adds the information to the context of the operation that no more replication steps are necessary. This means that the pre-aspects on the other container do not intercept the operation (steps 1.3.1.3.1 and 1.3.1.4.1). When the pre-aspect on container Node-1/C has successfully executed the *write* operation on the other containers it returns "OK" (section 4.2.2.1) and the original *write* operation is executed on the container. When that operation finishes the space pre-aspects has executed a write operation on all container replicas (step 1.4) transparent to the invoking application component (step 2).

Since the domain scenarios contain different types of information, each of them with different requirements on the replication strategy in sense of e.g. consistency, this kind of replication pattern is very suitable for e.g the information types where inconsistencies do not need to be resolved immediately or even at all, like in case of weather data. This also means that the replication strategy needs less overhead and is therefore more efficient than strategies which must keep the data distributed always in a consistent way. However, since the DHT replication strategy is only monitored to get informed about the changing location of replicas, the replication strategy implemented in aspects is totally independent. Therefore, aspects represent various kinds of replication and synchronization strategies transparent to applications and depend only on the type of information to be written into the container.

5.1.4.3. Coordination Support under Timing Restrictions

As described in the previous section and in section 3.3.1, there are various types of information and a lot of different stakeholders with diverse interests. Their aim is to

exchange information and coordinate each other to increase safety properties. A very simple coordination mechanism is the publish/subscribe paradigm. In safety-critical domains it is essential that each message is delivered exactly once, requiring so called durable delivery characteristics. However, in the described telematics scenarios it is not adequate at all to store all events a peer has subscribed for while the subscriber is off-line, and to deliver the events once the subscriber is reachable again. Beside concerns related to the storage of those events, the problem is that the subscriber would receive a large amount of data that has to be processed locally first in order to extract relevant information. Furthermore, mobile peers (vehicles) have only a few seconds of connectivity and very limited bandwidth, and therefore the number of messages to be transmitted is limited. Therefore, there is a risk that essential information, such as safety-critical ghost driver warnings, cannot be transmitted to the subscriber. If such messages are not forwarded to the peer on time humans lives may be jeopardized. Therefore, the safety risk grows with the amount of irrelevant or even outdated information delivered instead of important life-saving information.

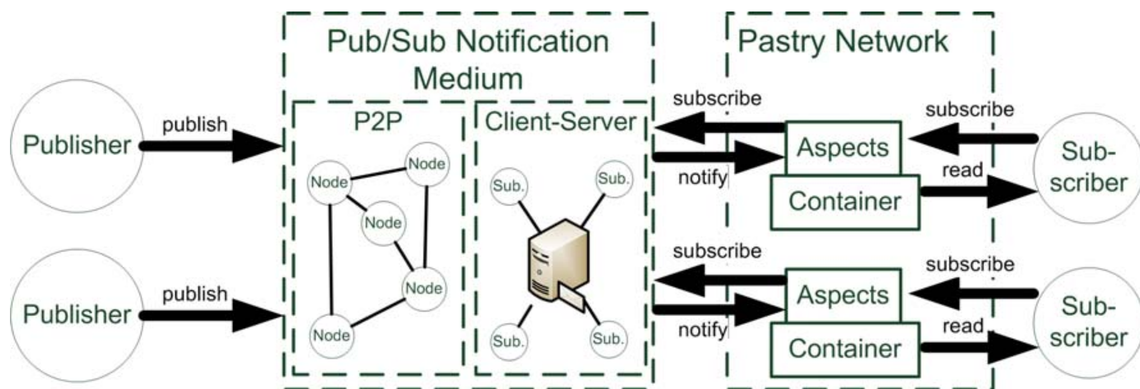

Figure 42: The operation of SBC and DHT concepts in a publish/subscribe scenario [125]

As shown in Figure 42, the proposed solution is to make use of DHT based distributed containers and of aspects acting as an intermediate-subscriber for events in the publish/subscribe system. It contains the advantages of

- a scalable lookup mechanisms for containers containing subscribed information
- pre-processing capabilities regarding delivered events to minimize the size of data and the time needed to transmit subscribed events to the real subscriber

As depicted in Figure 42 the original subscriber (e.g. a vehicle) places its intention in receiving events from publishers by deploying a container, installing an aspect, and making it public in the DHT overlay network in the same way as described in the previous sections. The aspect registers itself as a subscriber in the publish/subscribe system (whether P2P or client/server style) on behalf and according to the requirements of the original subscriber. From now on the aspect will, independent of the connectivity mode of the original subscriber, receive events which are then processed by other installed pre-aspects. The results of the final processing steps of the aspects are then

stored in the container. When the original subscriber re-establishes a connection to the network, it uses a read-selector to pick up the results from the container.

What kind of selector-type is used is completely up to the original subscriber and depends on the fact how the installed pre-Aspects work with incoming events. If the container is replicated, aspects are replicated as well. This means that the original subscriber is subscribed as often as many replicas of that container exist. This is necessary in order to avoid missing events in case one of the replicas, including the subscribed aspect, is off-line. The way how the replicated containers handle incoming events in order to stay consistent is up to the implementation of the deployed pre-aspects. An example of a possible replication strategy was described in the previous section. Replicas are either completely independent of each other and perform every operation as many times as replicas exists, or an incoming event is registered and not used for further processing until the result based on that event has been announced from a designated replica. The latter approach may be more efficient with respect to computational resources but requires knowledge about group coordination, thus increases the complexity of the aspect.

5.1.5. Summary

The integration of Distributed Hash Tables as transportation profiles into the communication and distribution category of the XVSM architecture enables the relocation and replication of containers in a distributed environment transparent to application components. Furthermore, the incorporation of the DHT concept adds Pastry's scalability, fault-tolerance, and self-organizing properties to XVSM containers, whereas aspects on containers transparently defined consistency strategies between them, as well as aggregation and filtering capabilities of events. The introduction of a space aspect allowed the application component to simulate writing and retrieving information into and from a container as it would have performed local operations.

5.2. *SWIS*

The aim of the SWIS project is to develop an "information sharing network" for the Air Traffic Management domain (Figure 13). The architecture of the SWIS platform shall facilitate a high availability of services in an environment with heterogeneous middleware infrastructures for the integration of distributed services with changing requirements regarding integration and network infrastructure.

5.2.1. Requirements concerning the Architecture

With respect to the domain scenario described in section 3.3.2 there are two categories of requirements. The first category refers to the heterogeneity of application services connected to the architecture of the SWIS platform. The platform has to fulfill requirements regarding

- integration of services over geographically large and distributed network groups
- heterogeneity of interconnected services and exchanged information
- integration of heterogeneous middleware technologies with different APIs, transportation capabilities, or network architecture styles

The second category deals with changing business goals and network infrastructures entailing the architecture of the SWIS platform to adapt accordingly, whereas adaptation should not require the reimplementation of SWIS architecture components. The category requires from the platform to cope with:

- Changing requirements regarding the way of communication between integrated services. This involves e.g., the change from a point-to-point way of communication to a publish/subscribe mode.
- Changing integration requirements. This involves clustering of services to so called virtual sender groups to increase availability of service information.
- Change of communication middleware due to e.g., lack of performance or scalability. This involves e.g., a change from a client/server approach to a P2P way of communication between nodes.
- Changing network node capabilities by adding extra network nodes physically clustered to improve availability of the platform.
- Changing communication requirements of integrated services regarding the underlying network infrastructure. This involves a change in the way of physical communication, like the usage of secure, wired communication paths only excluding communication via P2P networks.

Most importantly, the realization of the described changing requirements has to be performed not only transparently to the application services connected to the platform but also to software components of the SWIS architecture. From the point of view of this thesis the SWIS architecture components are application components representing the goals of a SWIS node. Since a SWIS node has to adapt due to changing integration requirements the architecture

5.2.2. Limitations of Related Technologies

System integration is the task to combine numerous different systems to appear as one big system. There are several levels at which system integration can be performed, but so far there is no standardized integration process that explains how to integrate systems in general [13] with the support of automated fully tools.

Typical integration solutions focus only on either heterogeneity on service level or heterogeneity on network level [121]. In order to cope with technological heterogeneity on service level a homogeneous middleware technology approach [92] is used for syntactical transformation between services, while the semantical heterogeneity of services is addressed by means of a common data schema [90]. Heterogeneity on network level may be addressed by using so called adapters transforming messages between each used combination of middleware technologies. However, in order to provide an effective continuous integration solution in this environment, both integration levels (i.e. service and network level) need to be addressed in a mutual way.

Traditional approaches for integration of business services can be categorized [40] according to the realized architecture (i.e. Hub and Spoke vs. distributed integration) and according integration logic (i.e. coupled vs. separated application and integration logic). In the following, a discussion about their advantages and limitations is given.

Application servers [92], a single hub integration architecture, are capable of interoperating through standardized protocols, but tightly couple integration logic and application logic together. As the name suggests a server based architecture style is used for integration and as such is inconvenient for scenarios with services widely distributed. Traditional EAI brokers [40] use a hub-and-spoke architecture; some of them are built upon application servers. The approach on the one hand has the benefit of centralized functions such as the management of business rules or routing knowledge, but on the other hand does not scale well across business unit or departmental boundaries, although it offers clear separations between application, integration and routing logic. Message-oriented Middleware [101] is capable of connecting applications in a loosely coupled manner but requires low-level application coding intertwining integration and application logic. The resulting effort and complexity of implementing an integration platform with the support for any kind of existing middleware technologies and protocols therefore is considerably high.

To enable transparent service integration, the Enterprise Service Bus (ESB) provides the infrastructure services for message exchange and routing as the infrastructure for Service Oriented Architecture (SOA) [178]. It provides a distributed integration platform and a clear separation of business logic and integration logic. It offers routing services to navigate the requests to the relevant service provider based on a routing path specification. Routing may be [40] itinerary-based, content-based, conditional-based defined manually [195] or dynamic [12]. Dynamic configuration focuses mainly on creating a route for a special business case. Using manual configuration, a system

integrator has to rely on his/her expertise, thus the high number of service interactions may get complex and the configuration error-prone. This may lead to routes that are configured in a way in which their influence on other business interactions is not fully known. Additionally, dynamic configuration may not cope with e.g. node failures fast enough due to missing routing alternatives, therefore possibly violating time-restricted non-functional business service requirements.

5.2.3. Description of the proposed Architecture

As described in section 3.3.2 the SWIS network consists of various intermediate nodes forwarding information from one application service to another. Beside heterogeneous network infrastructures an intermediate node may be clustered by means of shadow nodes, or may form a virtual coordination group for particular services to ensure high data availability.

In order to demonstrate the capabilities of the SBC platform with respect to agility, in the following sections it is explained how the SWIS architecture copes with changing business integration requirements, and how the architecture of a SWIS node is realized that is transferable into a shadow node or part of a virtual sender group.

5.2.3.1. Handling changing Business Requirements

Today companies and organizations operate in a highly complex environment requiring well-defined but flexible means for communication and cooperation that can be easily adapted to potentially frequently changing business processes. One of the major challenges in safety-critical environments like the Air Traffic Management (ATM) domain is the fulfillment of all functional and nonfunctional requirements of business services to be interconnected. In such safety-critical environments, both semantically and technologically heterogeneous services, which were originally not designed for flexible integration, need to be integrated on top of heterogeneous middleware infrastructures in a deterministic and static, but also fault-tolerant manner.

The properties of a safety-critical domain [102] describe clearly how decision making has to be performed. For each decision made, a response has to be pointed out, and based on the same internal states the same decision has to result in the same output. Additionally, any missing but needed information may lead to catastrophic scenarios. Therefore, an error-prone and time-consuming manual configuration of the system is forbidden, while dynamic configuration of the system is prohibited due to the possibility of a non-deterministic outcome of the decision process.

To tackle the manual configuration problem we use the model-driven system configuration (MDSC) [152] approach that automatically derives system integration configurations from changed business requirements for the SWIS platform. The MDSC approach is similar to the Model Driven Architecture (MDA) concept which major goal is to separate system functionality specification and implementation [209].

Using the so called Computation Independent Model (CIM) the MDA framework can be used to construct the models describing system requirements and behavior in a formal way, like e.g. by using UML. The separation of system functionality and implementation specifications is modeled in the Platform Independent Model (PIM), which is refined out the CIM, normally by hand. The main functionality of the PIM is to specify system structure and behavior independently of the platform it is deployed on. The separation of system functionality on a specific technology platform is described in the Platform Specific Model (PSM), which is the result of the PIM transformation to the target platform. The advantages [110] of the MDA framework are (1) automated generation of results improving productivity, development time, and cost; (2) focus on the creation of conceptual models rather than on logical and technical details; (3) resuse of transformations; (4) adaptations due to changes of the target platform concern the PIM only; and (5) new requirements defined in the CIM are passed to PIM and PSM immediately and therefore changes are reflected automatically [209]. The main difference between MDA and MDSC is that the result of the transformation is systems configuration models rather than code.

In the MDSC approach, based on requirement (e.g., accuracy of delivered information) and capability (e.g., frequency of delivery) models of business services and on the capabilities (e.g., bandwidth, latency) of the components of the heterogeneous network infrastructures [154], a logical solution model which represents the set of suitable integration partners is derived automatically [155]. This logical solution model then is transformed into the technical solution model, representing the specific integration configuration for the underlying SWIS platform.

Figure 43 shows the approach for configuring the SWIS platform based on the Model-driven Architecture [148] concept. The approach automatically derives integration configurations from the semantic description of services. Beside the service capabilities and requirements [155], the components (e.g., network nodes and links) of the heterogeneous network infrastructures and their capabilities (e.g., bandwidth, latency, security) are explicitly modeled. Since this thesis concentrates on agility aspects of software architectures rather than on semantic integration, details regarding semantic modeling, matching of ontologies, and reasoning can be found in [23, 152, 153, 156] and are no treated here further.

For each service to be integrated, the subject matter expert responsible for the particular system describes the messages which are either provided or consumed by the services and their contracts (i.e. quality of service) regarding the integration with other services.

In addition, the participating services define extra requirements (e.g., security constraints) regarding the underlying heterogeneous network infrastructure [154]. The described semantic models are used to automatically derive the set of possible integration partners using ontology-based reasoning [154, 157]. Services which have been derived for a successful integration create a so called *collaboration*. The collaboration specifies the services that provide information and services that consume that information including the properties of the collaboration (e.g., security aspects). The result of this step is the so called Logical Solution Model.

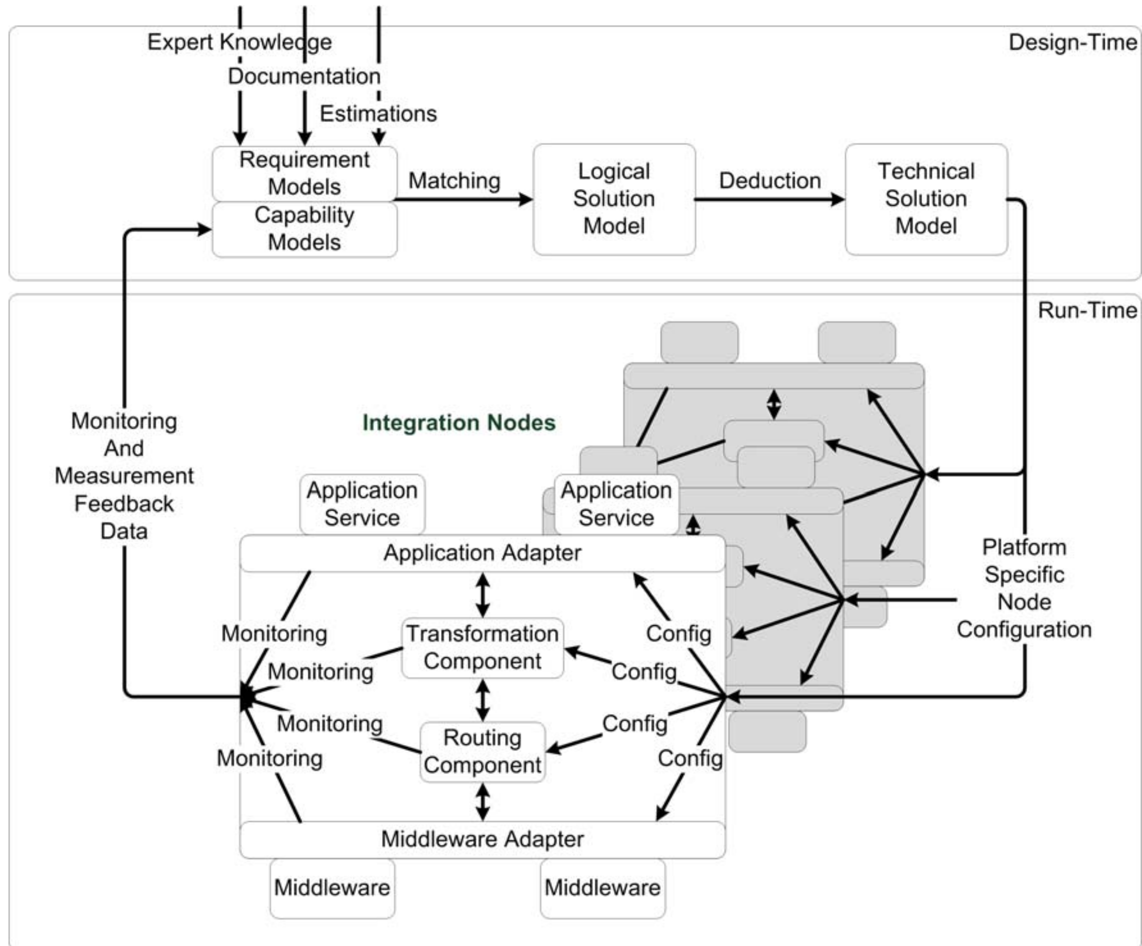

Figure 43: Configuration of the SWIS platform

Based on the Logical Solution Model, the Technical Solution Model for each SWIS node is generated automatically [152] using evolutionary algorithms [119, 120] by taking into account service infrastructure requirements and infrastructure capabilities. While the Logical Solution Model describes a set of collaboration partners, the Technical Solution Model represents the mapping of collaboration specifications onto the physical network infrastructure and thus the proper functionality of each SWS node. The Technical Solution Model is an XML configuration which is deployed on and interpreted by each SWIS node. The major components of the model are: routing information, transformation instructions, and extension modules.

Routing Information

Routing information represents the routing table that is used by each SWIS node to forward messages correctly. Routing tables specify where certain received messages belonging to a specific collaboration should be forwarded to. Based on the type of the message the configuration specifies whether the message is forwarded to another SWIS node or to a local service. If the message has to be forwarded to another SWIS node the configuration specifies the concrete extension module to be used. The routing table specifies several routing targets for a specific collaboration (so called "backup routes"), which are used in case of unavailability of the original target SWIS node. In case the message has to be sent to a local service, the configuration specifies the concrete service.

Transformation Instructions

Transformation instructions define how messages originating from business services must be manipulated before sending them to other business services. Instructions specify how to transform messages from one specific structure to another. The instructions specify one of the following rules:

- do nothing with the message
- change message data types, convert e.g. float numbers to integer
- split a message into several messages
- merge several messages of different type into a single message
- replace or enrich certain information of a message with or by another one by calling external services

Extension Modules

An extension module describes the semantics of a component rather than the specific configuration a component. This allows replacing specific aspects of the SWIS architecture with different component implementations but with the same semantics. There are four types of extension modules which describe non-functional constraints regarding the SWIS architecture:

Communication Module Specifications describe the semantics of communication between two SWIS nodes. It refers to requirements like security or failure management.

Application Module Specifications describe the components to be used allowing communication between the SWIS node and the connected service.

Clustering Module Specifications describes the semantics of strategies (e.g., replication, consistency) to be used when a SWIS node has to work with shadow nodes.

Virtual Group Module Specifications describes the semantics of coordination strategies to be used when a SWIS node belonging to a virtual group has to agree with other SWIS nodes on a particular member of that group to represent the group permitted to forward messages.

Since the MDSC derives only the semantics of a component (i.e. the requirements the implementation has to fulfill), an additional MDSC processing step is required that is capable of matching the derived requirements with the capabilities of already implemented components stored in a repository. In the SWIS project we implemented this processing step in a static way by pre-specifying the concrete component implementations to be used. An approach towards an automated processing step is to use the same mechanisms as performed for matching service capabilities and requirements. The MDSC is able to derive the semantics of a component (i.e. component requirements) which need to be matched with components in the repository (i.e. component capabilities).

5.2.3.2. SWIS Node Architecture

Figure 44 shows the main components of a SWIS node where the XVSM framework is used to enable the intercommunication between its various components. Each component has its own container used as an inbox for messages to be processed. The components communicate with each other by writing a message into the container of the component. The behavior of the components is specified by means of the derived Technical Solution Model. The components of a SWIS node are [152]: Application Adapter, Transformation Component, Routing Component, Middleware Adapter.

Application Services (AS) are application components which send messages to or receive messages from other application components. The SWIS node is responsible to deliver such messages from the source AS to the destination AS.

ASs are connected to the SWIS node by means of an **Application Adapter** (AA). An AA represents a gateway capable of communicating with both the application and XVSM and has been configured according to the application module specification.

The **Transformation Component** executes transformation instructions specifying how to handle certain message types [154-156]. The instructions have been derived from requirement and capability models describing the services. The component can change message data types, split messages into several segments, merge different segments into a message, replace or enrich certain information of a message, or perform any combination of the described possibilities.

A **Routing Component** contains routing information specifying where to forward a message to. A message can be either forwarded to a local service via the Transformation Component or to another SWIS node via the Middleware Adapter. If a message cannot

be sent to the specified SWIS node, "backup" routes allow the component to react on changing network conditions quickly.

The **Middleware Adapter** (MA) is a transportation profile that abstracts a middleware technology and that is configured according to the communication module specification.

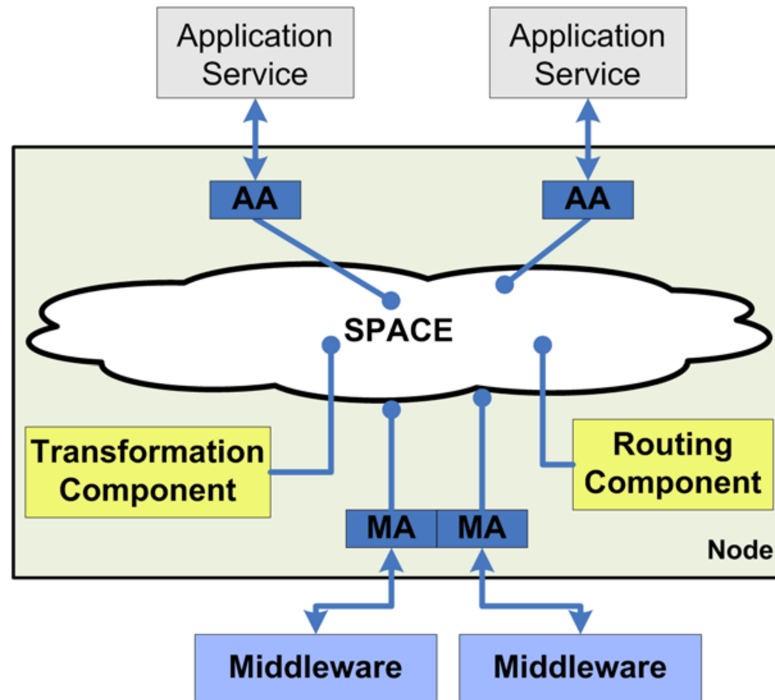

Figure 44: Components of a SWIS Node

The **Middleware** component represents a concrete middleware technology capable of transmitting data between two SWIS nodes.

5.2.3.3. SWIS Node operation

The main focus of this section is to explain the flow of a message between two collaborating services (Figure 45). In this scenario, the collaboration consists of a sender service (Service 1-S1) and a receiver service (Service 2-S2).

If S1 sends a message to its collaboration partner S2, the processing of that message is done sequentially. S1 writes the message via AA into the container C1 with FIFO coordinator. The Transformation Component performs blocking take operations removing the message from that container. Then it executes transformation operations on the message according to the specified transformation instructions.

Depending on the transformation instruction, a number of messages are written into the container C2 with FIFO coordinator that is used by the Routing Component. Messages in that container are removed and processed. The Routing Component looks up its

configured routing table and finds out where the message has to be sent. The routing table does not specify the exact middleware technology but the container abstracting the communication and transportation channel of the SWIS node where the message should be transmitted.

When the Routing Component writes the message into the specified container VC1, a pre-aspect intercepts the message. That pre-aspect invokes the transportation profile of the middleware technology responsible for transmitting the written entry (i.e. the message sent by S1) to the specified destination. The MA is responsible for forwarding the request to the middleware technology it represents. In case the called middleware technology returns an error due to network failures the pre-aspect aborts the operation and returns "NOK" (section 4.2.2.1) to the Routing Component. The Routing Component looks up an alternative route and writes the message into the container specified in its routing table.

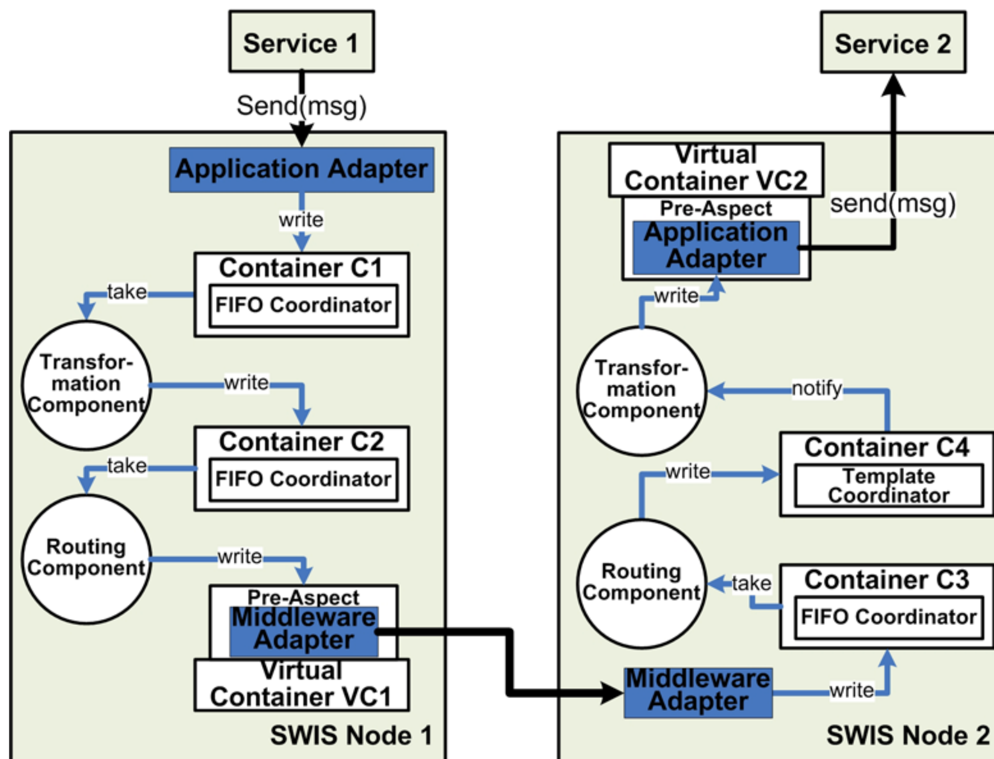

Figure 45: Processing of messages in a SWIS node

On SWIS node 2 the MA receives the message and immediatly writes it into container C3 of the Routing Component. The Routing Component takes the entry out of the container, analyzes it according to its routing table, realizes that it has to be forwarded to a service and writes the message into container C4. The Routing Component also tags the message with the information specifying a container (i.e. VC2) by which the Transformation Component addresses the targeted service.

At this point it has to be mentioned that the containers C4 and C2 have to be separated. While for the Routing Component it is irrelevant whether it processes incoming messages (i.e. an entry written by an MA) or outgoing messages (i.e. an entry written by the Transformation Component), the Transformation Component has to fulfill different responsibilities (e.g., split vs. merge of messages) requiring the support for separation of concerns by two separate containers. In contrast to the operations executed so far, the Transformation Component does not perform blocking take operations on C4, but registers subscriptions using consuming notifications which push incoming messages to the component when the subscription triggers. A subscription is configured according to the specified transformation instructions and takes into account the identifier of the collaboration and the message types. This allows the Transformation Component to be notified when several messages have arrived which need to be merged according to the transformation instruction. If the component used take operations, it would have to manage the temporal storage messages part of a merge instruction resulting in a complex component implementation.

Once the transformation of messages has been finished, the Transformation Component writes the message into the container that has been specified by the Routing Component. That container (e.g., VC2) is a virtual container hosting a pre-aspect. The aspect intercepts the operation and forwards it to its Application Adapter which send the message to the service it represents.

5.2.3.4. Shadow Nodes

Shadow nodes belong to a particular SWIS node and are an exact copy of it. In Figure 15 and Figure 16 there are certain SWIS nodes shown, but in reality there may be far more, since shadow nodes are hidden from the outside. A SWIS node and several shadow nodes represent a Clustered SWIS node. Clustering refers to the definition of tightly coupling SWIS nodes that work together closely so that they seem to be a single SWIS node. Clustering of SWIS nodes is introduced to improve availability, and not to increase throughput.

A clustered SWIS node consists of a main SWIS node and several shadow nodes. A shadow node serves as a backup SWIS node in case the main one fails and is configured the same way as the main SWIS node. Shadow nodes are only existing as backup and do neither perform any calculations nor other execution processes until the main SWIS node has crashed.

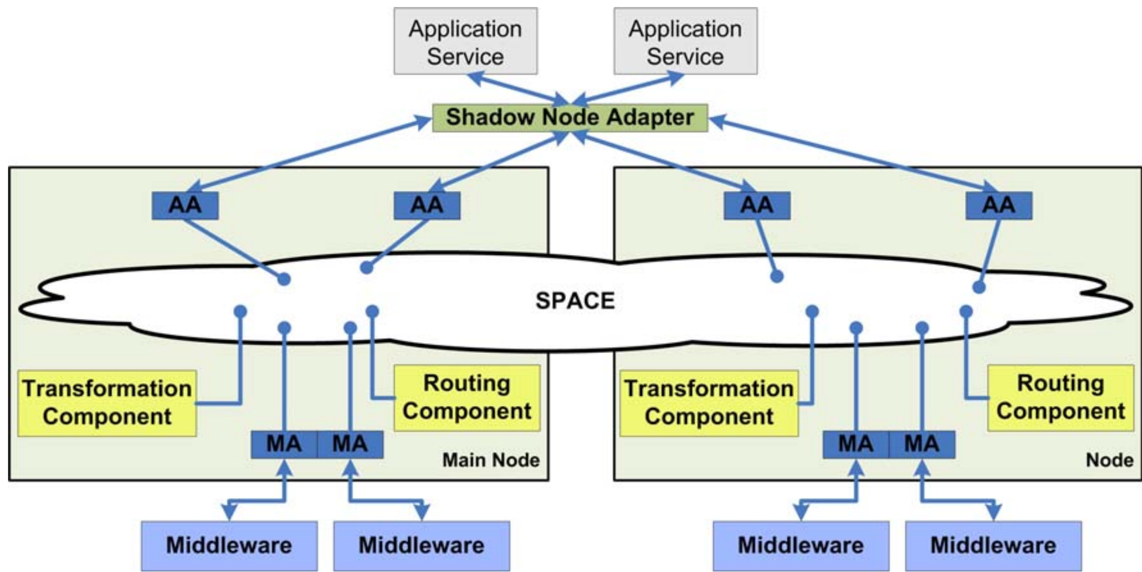

Figure 46: Components of a Shadow Node

The architecture respectively the main idea of the cluster is shown in Figure 46. The difference to a normal SWIS node is that the used space is extended over the entire cluster creating a "shared memory" over the single SWIS nodes and the introduction of a so called Shadow Node Adapter (SNA).

The SNA adapter is a physical component that analyses data packages at IP-level. The Technical Solution Model provides the SNA adapter with information about how many shadow nodes the cluster has, which of them is the main one, and in which prioritized order the shadow nodes are running. This means that if the main SWIS node fails, the backup shadow node is already known avoiding elections for a new leader.

Extending the space over many nodes is done by distributing each container of the SWIS node to all shadow nodes and keeping them synchronized. With respect to the process of deriving the Technical Solution Model this means that each container (C1, C2, C3, and C4) of Figure 45 is equipped with a pre-aspect according to the XVSM distribution category (section 4.2.4.4) transparent to the application components (i.e. Transformation Component and Routing Component). The pre-aspect is an Extension Module and represents the semantic meaning of the derived Cluster Module Specification while the hosted component implementation is responsible for keeping the replicated containers synchronized (Figure 47) according to the derived specification.

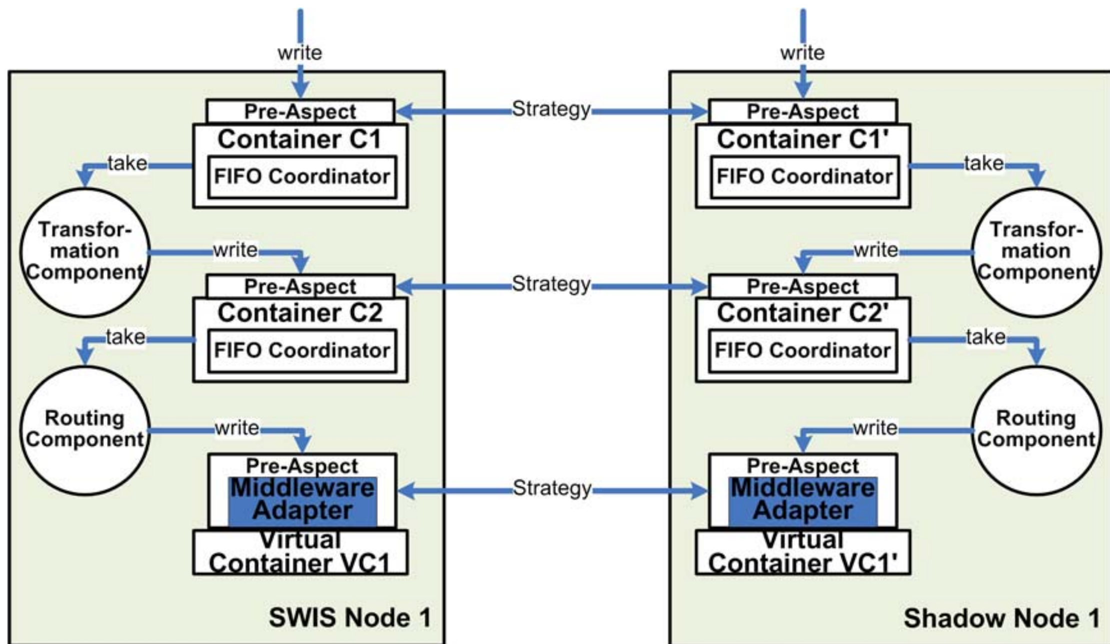

Figure 47: Distribution category of SWIS Node 1

As an example, a hosted component may be an XVSM specific Extension Profile (section 4.1.2) or vendor specific Java Message Service (JMS) [150, 190] implementations supporting clustered queues.

Figure 47 shows a SWIS node and a shadow node, where each container C of the SWIS node has a counterpart C' at the shadow node. Pre-aspects installed on each container make sure that messages placed into C1 or C2 are distributed to C1' and C2' as well. Pre-aspects installed at the virtual container VC1 and VC1' have an additional responsibility. They also have to make sure that the written message has been really sent to the destination node. If SWIS node 1 crashes during transmission then VC1' is in charge of resubmitting the message. An example for a component realizing the cluster module specification regarding the execution of synchronization and replication of incoming messages is described in section 5.1.4.2.

5.2.3.5. Virtual Sender Groups

A Virtual Sender Group is a group of services producing messages of the same type. The main characteristic of that group is that only the produced message of a single application service may really leave the group. In some cases services cannot be stopped from creating and sending messages (e.g., radar stations), but SWIS has to be able to prevent the message from leaving all the SWIS nodes but one.

The problem of virtual sender groups reveals the need for coordination between the participating members of that group. To keep the abstraction interface reduced to the

methods *write* and *take* and the implementation of the platform less complex, it has been avoided to add an additional component to the SWIS node architecture responsible for group communication only. Therefore the configuration of a SWIS node with virtual sender group characteristics is similar to the configuration of clustered nodes.

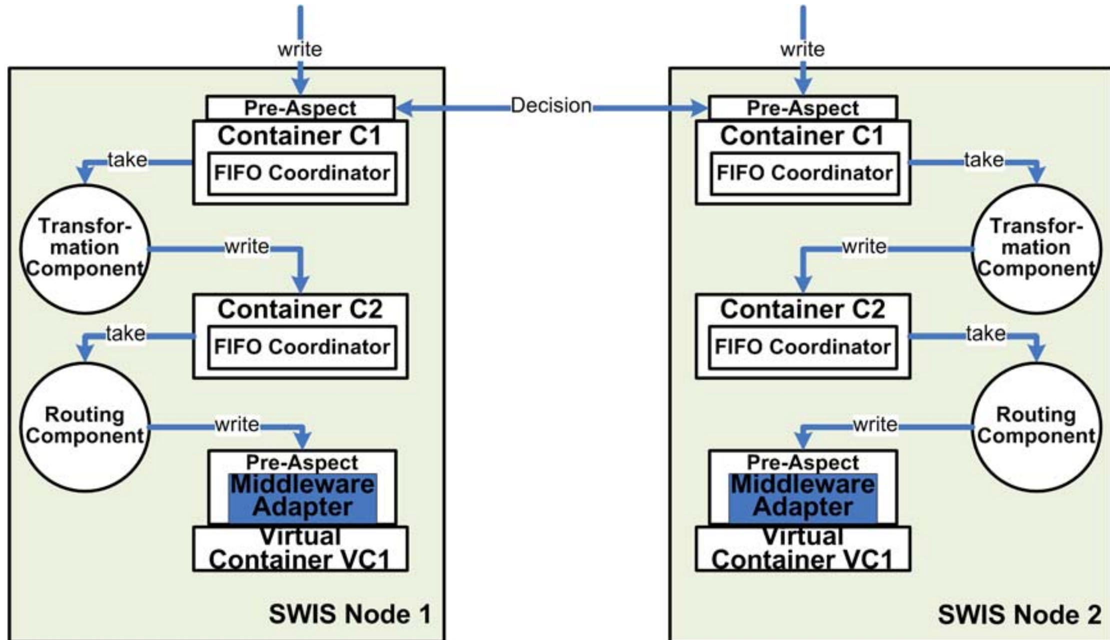

Figure 48: Reaching group decisions in the distribution category of XVSM

If the Technical Solution Model requires the implementation of Extension Modules based on Virtual Group Module Specifications, then pre-aspects representing the semantics of the specified configuration are installed on C1 containers of each SWIS node. The pre-aspect intercepts incoming messages and forwards it to the component implementing the actual specification for coordinating a group decision making process [185]. When the aspect receives a message from an AS that is a member of the virtual sender group, it withholds the message until the group has reached a decision. Once a decision has been reached, the message is either processed by the other SWIS components or discarded.

5.2.4. Summary

The aim of this section was to show that the SWIS architecture is capable of standing changing business requirements with no effects on components part of the software architecture. The integration of heterogeneous application components over heterogeneous network infrastructures is achieved both during design time and runtime. Processing steps during design time derive a configuration out of capability and requirement models. The SWIS architecture performs integration according to this configuration and thus enables integration of application components during runtime.

Every time characteristic of application components or network infrastructures change, a new configuration is calculated requiring from the architecture to adapt accordingly. In this project, the XVSM container concept was used to abstract concrete resources enabling communication between architectural components. Starting with a simple requirement, the FIFO coordinator was used in the container to simulate queue behavior. However, the abstraction enables the transparent switch to other vendor-specific implementations as long as those components reflect the given communication semantic (i.e. queue). If due to changing business integration requirements or network infrastructure capabilities the transparent introduction of clustering capabilities or group agreement policies has to be enabled, XVSM's extension profiles realize these strategies and can be added to a container without affecting the architectural components. The result is that XVSM concepts enable programmable connectors.

The section described that the SBC architectural style and the concepts of XVSM allow the adaptation of new requirements transparent to architecture components. Finally, it has to be mentioned that the aspect of deployment was not part of the project.

5.3. *SAW*

The SAW research project investigates coordination and recovery capabilities of software agents (i.e. application components) in the production automation domain as an approach for reintegration of failed software agents into the coordination process.

5.3.1. Requirements concerning the Architecture

Agents control the underlying machinery they are representing and are responsible for interacting with the environment. Therefore, they need to cooperate with other agents in order to satisfy business requirements [147], such as the optimal usage of the production system in order to allow just-in-time delivery and cause minimal production costs. For instance, agents determine the optimal transportation path of pallets from one production machinery to the next [146, 224].

However, as any other distributed system, such systems are prone to failures as well and agents may crash due to hardware or software failures. This implies that the underlying machinery is not under control any more, resulting either in an immediate halt of that part of the production system or in an immediate execution of certain techniques to avoid further undesirable actions.

The downtime of a part of the production system in case an agent crashes is critical, since costs can rise and production be delayed. It is important that the time needed for a recovered agent to achieve its "optimal" state for production is as minimal as possible in

order to allow an overall optimal (i.e. as intended in specifications) usage of the production system after recovery. Since only one agent at a time is capable of controlling the underlying machinery, running several replicated agents of the same type is not possible. Thus, a new instance of the crashed agent has to be executed. The problem with this technique is that the newly started agent is not aware of the circumstances in its new environment, requiring time consuming and extensive message exchange with neighboring agents to update its view on the system and to work properly. Therefore, more sophisticated system recovery techniques have to be considered: "*A system recovers correctly if its internal state is consistent with the observable behaviour of the system before the failure.*" [133, 214]. However, in the production automation a recovery from software faults is not sufficient enough, since the environment, the crashed agent is not part of any more, may have changed while a new instance of the agent is being started. Therefore, the state information at the time of recovery should already reflect occurred changes in the environment. The addressed issue is a sub-problem of the agent recovery problem, namely the assimilation problem [15]. This problem describes the issue of how an agent is capable of "catching up" with the external environment, by reaching a foregone considered state, in case the agent had not crashed.

5.3.2. Limitations of Related Technologies

Rollback-recovery protocols [64] are key strategies to achieve fault-tolerance and have been developed in order to increase reliability and availability of distributed systems. Those protocols can be classified into log-based and checkpoint-based protocols.

Log-based recovery [6] uses a message log for each agent periodically recording its local state and log the messages it received after having recorded that state. Upon failure the state of the agent can be reincarnated [15] through the playback of the logged messages. However, nondeterministic events have to be stored as well in order to ensure that the agents' behavior is the same with respect to other agents. Additionally, it needs a lot of storage and processing for reincarnation.

In case of coordinated checkpoints [132] consistent set of checkpoints form a recovery line so that all agents can roll back to a consistent global state. However, in case of checkpoints it is difficult to roll back to a consistent state. Actions already performed by the underlying machines, like the assembly of product parts or painting, cannot be undone. Furthermore, log-based recovery does not store messages while the crashed agent is down. This implies that the agent, once up and running again, is capable of restoring state information, but runs the risk of working with out-of-date information. This means that the agent is on the one hand capable of behaving correctly with respect to its objectives, but on the other hand it may use information that is in conflict with the

current state of the production system. Thus, the agent would have to analyze the environment any way to catch up and work within the requirements.

5.3.3. Description of the proposed Architecture

This section describes the architecture of the SAW applications in four steps: initialization, preparations for recovery, interacting in the presence of failures, and interaction after recovery. The description of the proposed architecture uses the agent's routing tables as an example to explain the four steps. As shown in the following sections, the proposed architecture uses a strategy that distributes information among existing software agents. The reason is the intent of the community [41, 97] to avoid central components which can be seen as a single point of failure.

5.3.3.1. Initialization

Each agent in the production system has its dedicated container, for storing data structures. For instance, a conveyor belt agent uses a container with FIFO coordinator while a crossing agent uses a coordinator optimized for routing tables. As shown in Figure 49 there are four crossing agents each having access to their containers called C1, C2, C3, and C4.

When the system is started the very first time, all existing CAs (crossing agents) exchange information about incoming and outgoing CBAs (conveyor belt agents) and the costs of each conveyor belt to get informed about the environment they are running in. This information is collected, analyzed, and stored in their containers. Table 2 shows four containers. Each of them stores routes to a destination prioritized according to the costs of the connected conveyor belts. For instance, the "cheapest" route from Agent 1 (i.e. agent C1 belongs to) to docking station DS1 is accessed if the incoming pallet is forwarded to the outgoing conveyor OC_0 going to crossing agent C2. OC_x refers to the container that represents the outgoing conveyor X.

Beginning	Costs	Route
C1	OC_0(40)	C2,C4
	OC_1(50)	C3,C4
C2	OC_0(30)	C4
C3	OC_0(40)	C4
C4	OC_0(10)	-

Table 2: Costs and routes from every container to destination DS1

5.3.3.2. Preparations for recovery

For recovery of agent specific information for efficient reintegration of a failed agent into existing coordination processes, e.g., the replication strategy as described in section 5.1.4.2 can be used too. There, containers are replicated transparently to the agents in combination with a DHT overlay network for an efficient and fault-tolerant lookup of containers. This section also details that by using aspects it is possible to deploy several replication strategies to keep containers consistent. In contrast to the RealSafe project, aspects in SAW are responsible for replication by executing a multicast protocol, called JGroups[17].

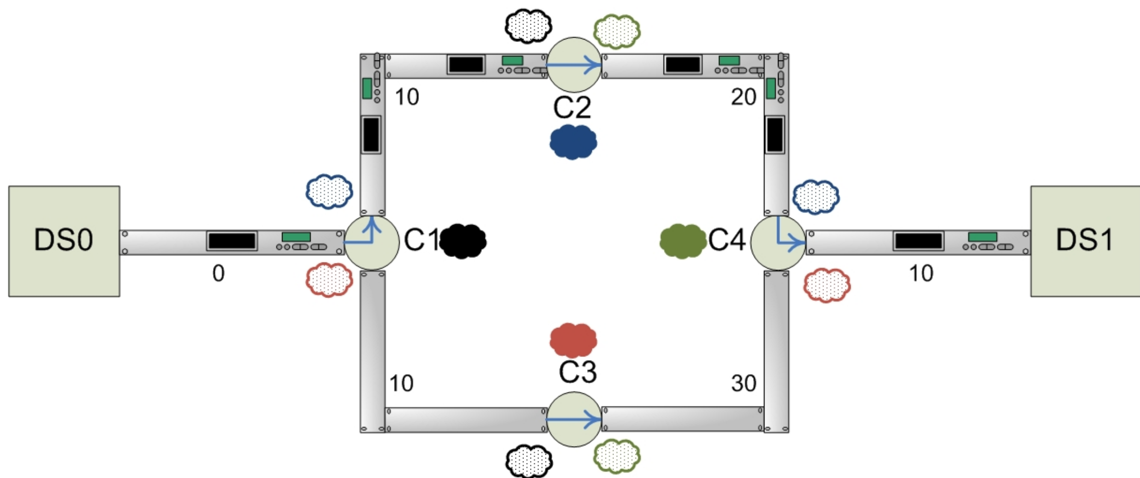

Figure 49: Data structures for storing routing tables

As shown in Figure 49 each of the four containers (black, red, green, and blue) is replicated three times. The original containers are fully painted, while replicas are shaded. Replicas are placed at the nodes of the next neighbors.

Furthermore, the hash-function of the DHT concept has been altered in a manner, which allows containers to be stored with respect to the structure of the production system. This means that containers are located in different areas of the production system in order to increase availability of data (Figure 50).

---

[17] http://www.jgroups.org/

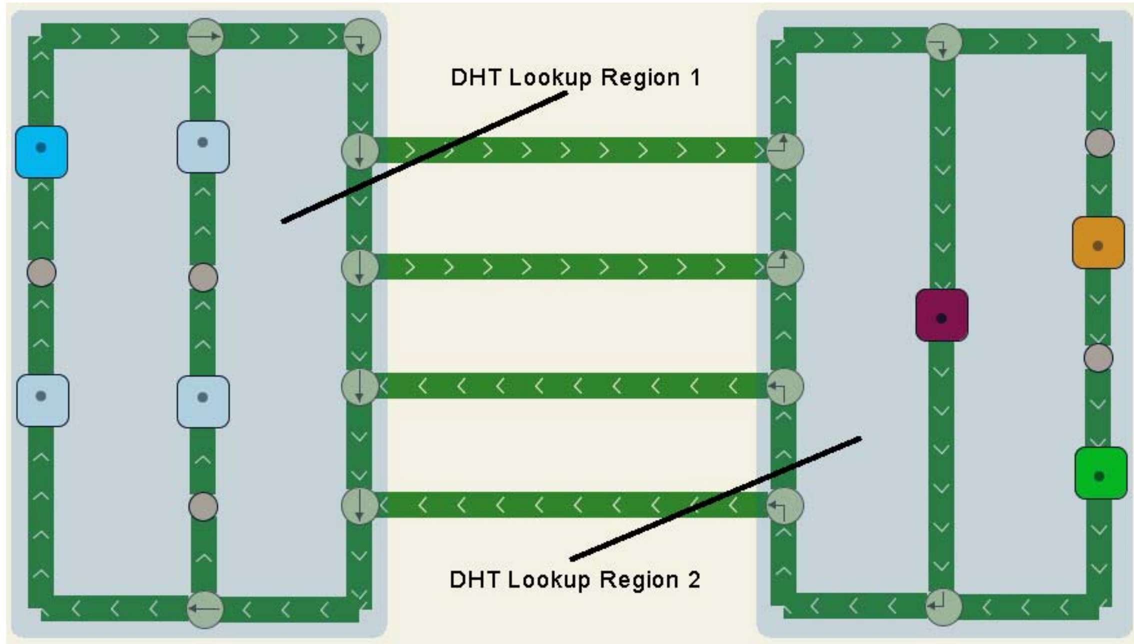

Figure 50: Production automation system split into several DHT lookup areas

Figure 50 shows the structure of a production system which is divided into two DHT lookup regions. Instead of referring to geographic locations in the DHT function, the structure of the production system is taken, and separated into regions allowing container replicas to be placed there.

5.3.3.3. Interacting in the presence of failure

Figure 51 shows lookup areas, replicated containers, and the integration of XVSM into agents. The figure presents triple replicated containers for four exemplary agents located in four overlapping DHT lookup areas, also called replication clusters.

In case there is a change in the described layout, like due to the failure of crossing C2 in Figure 49, the first (adjacent) component that discovers the failure updates its local routing table by disabling that specific route. The routing information to the failed agent (i.e. the route from C1 over $OC_{0)}$ in the container is disabled by setting its costs to a negative value). However, post-aspects on that container realize the executed change on the routing table and trigger updates of routing tables previous to crossing C1 transparent to the owner of the container. Therefore, any pallet passed along C1 with DS1 as destination, the path over OC_1 will be assigned to.

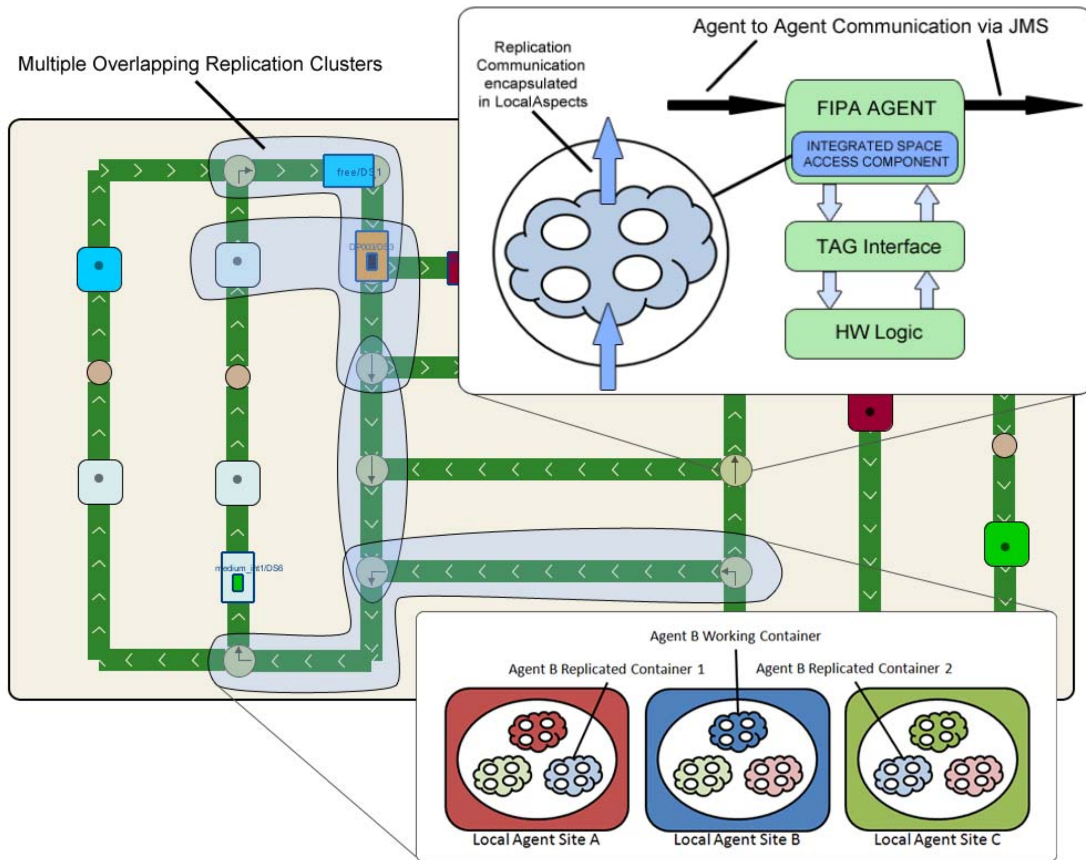

Figure 51: Production agents with triple replicated containers using various DHT lookup areas (replication clusters)

This means that aspects deployed on containers analyze incoming events and based on the type of the event they manipulate the container of other agents transparently to agents they belong to. The concept allows creating a shared container with a global view on the current state of the production system.

Since agent logic and routing information is separated, the state information of the failed agent can be updated as well. This means that in case another agent in the system has crashed, aspects can still update the routing table of the already crashed agent. Once the crashed agent has recovered, it can immediately access its up-to-date routing table by retrieving its container.

5.3.4. Summary

This section described the architecture of a system supporting zero-delay recovery of agents. The proposed architecture uses XVSM concepts which abstract the distribution of containers (distribution category) containing routing information. Updates on routing

tables are automatically disseminated to other agents (i.e. to other containers containing the agent's routing table) by means of aspects (organization category).

Chapter 6

6. Evaluation and Discussion

This chapter presents the results of the evaluation of the Space-Based Computing architectural style and discusses these results with regard to the specified research issues (see section 3.1). The first section describes the evaluation of the RealSafe, SWIS, and SAW application scenarios. Furthermore, it presents evaluation results gained by conducting studies regarding the usability of the SBC architectural style based on students' work at the Vienna University of Technology. The second section gives a general discussion of the benefits and limitations of the SBC architectural style.

6.1. *Evaluation of Application Scenarios*

This section presents the evaluation results of the RealSafe, SWIS, and SAW application scenarios, as well as of the conducted study.

6.1.1. RealSafe Application Scenario

The RealSafe project integrates the SBC approach and the DHT concept for the distributed storage and dissemination of structured and short lived data. Additionally, it facilitates coordination of stakeholders under communication disruptive conditions.

The evaluation results represents a contribution to support the thesis' claims in section 3.1.1 and 3.1.2 regarding the efficient coordination of application components (C1.2), continuous coordination support for recovered application components (C1.3), a flexible architecture (C2.1), and reduction of complexity of application components (C1.4).

The section investigates the claims from four different perspectives. The first perspective discusses the effectiveness and efficiency of the integrated approach. The implemented prototype is effective if it is capable of providing at least the same functionality regarding information retrieval and dissemination as a pure DHT approach. The prototype is efficient if a vehicle driver is capable of retrieving structured data in less amount of time than a DHT supported solution and if a publisher of information is capable of disseminating more information into the system than a vehicle driver can retrieve via an RSU. The second perspective deals with the question how flexible the architecture is in case of changing coordination requirements. The SBC architectural style is effective if application components do not need to be changed in

the process of change. The third perspective deals with continuous coordination support. The architecture is efficient if the number of relevant transmitted messages is higher than the number of messages transmitted via traditional publish/subscribe systems in the context of mobile subscribers. The fourth perspective refers to the complexity of the solution. The approach is efficient if complexity is reduced, i.e. the number of architectural concepts remains equal and the number of processing steps decreases in comparison to traditional solutions and architectures.

6.1.1.1. Performance of Integrating SBC and DHT

In order to answer the question regarding the effectiveness and efficiency of the integrated solution we have implemented a prototype and performed tests. As defined, the aspect of efficiency measurement is divided into two parts: information retrieval and dissemination.

Efficiency of Information Retrieval

In the conducted tests a specific message belonging to a specific road segment that is responsible for a specific region has to be retrieved out of a large number of messages also meant for that road segment.

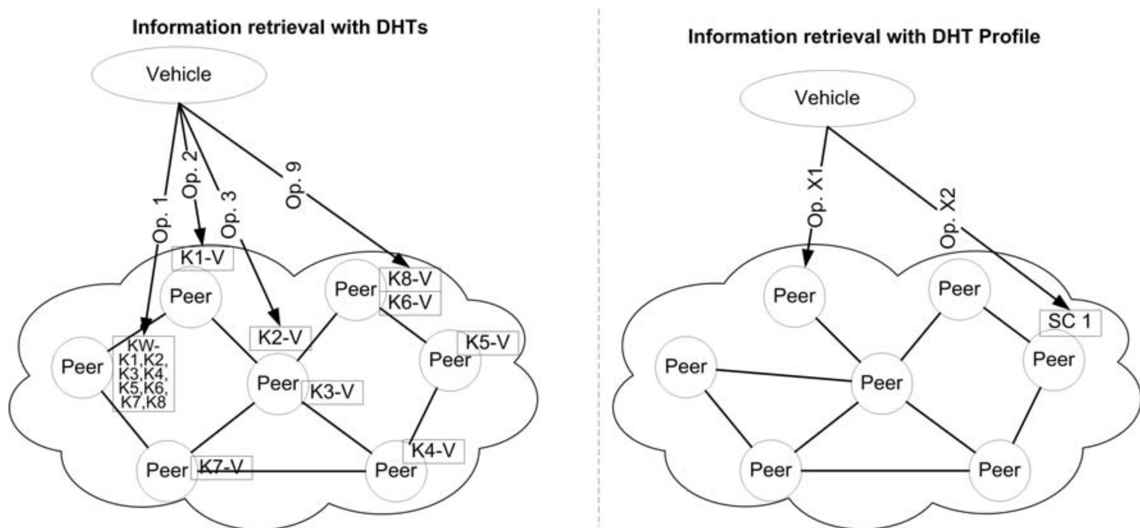

Figure 52: Complexity comparison in case of retrieving [123]

Figure 52 shows a diagram with the sequence of operations for the test case. On the left side a plain DHT solution is illustrated and on the right hand side the integrated solution.

Since a DHT is not capable of querying for a specific key out of a set of keys using a template (section 5.1.3.2), a well-known key (KW) has to be introduced into the system. That KW represents a specific road segment and its value stores every key by means of

region-specific information can be retrieved. The well-known key is needed since a vehicle driver cannot know the keys beforehand. The well-known key is a starting point for his/her search.

In the plain-DHT setup, the vehicle client retrieves the value of the KW corresponding to the road segment name. This means that the driver retrieves the keys containing region specific messages. In a second step, the individual keys are used to retrieve their values (i.e. messages). However, the test case defines that a specific message that matches particular interests has to be found. This means that every retrieved value of each key in the value of the KW has to be evaluated. In the best case, the first key that is used for information retrieval already contains the message that matches the interest. In the worst case every single key has to be used, meaning that the very last key contains the value that matches the interest. Listing 1 shows the retrieval algorithm.

Nr.	Operation
1.	`values = lookup(KW); // lookup value of well-known key`
	`// get values of received keys`
2.	`while ((key = nextKey(values)) != FALSE) {`
3.	`   info = lookup(key);`
4.	`   if (info.matches(interest)`
5.	`      Continue;`
6.	`}`

Listing 1: Retrieving region-specific information with a well-known key pair.

In the DHT and XVSM integrated setup, the DHT entry points to the container SC1 using a Template Coordinator that contains messages for a specific region. This implementation requires the execution of only two operations: the first operation retrieves the container reference URI (CREF), e.g., by means of a well-known key (CNAME) via the DHT. The second operation retrieves the message from the container that matches the interest of the vehicle client by specifying the template in the Linda Selector (Listing 2).

Nr.	Operation
1.	`CREF = lookup(CNAME); // get Container reference`
	`// get all entries interested in`
2.	`read(CREF, LindaSelector(template))`

Listing 2: Retrieving region-specific information with a well-known key pair.

This means that the integrated solution is effective since it is capable of providing region specific information, as the plan-DHT solution.

In order to quantify the efficiency of the proposed approach, we measured the average time needed to run the test case out of ten runs. In the performance test case, the task is to retrieve a message out of 10, 100, or 1000 messages belonging to the same road segment based on a network with between 10 and 210 peers and measure the time needed to do so. We used the WINZIG Grid [118] at the Vienna University of Technology to run our test cases. The Grid consists of 300 standard desktops clustered in groups of 30 machines. The groups are interconnected with high-speed switches organized hierarchically.

Configuration	Peers	10 Entries	100 Entries	1000 Entries
Plain DHT	2	0.3	4.4	61.3
Plain DHT	10	283	2,485	24,529
Plain DHT	60	263	2,455	24,126
Plain DHT	120	277	2,586	24,153
Plain DHT	180	264	2,491	24,252
Plain DHT	210	263	2,613	24,098
DHT+XVSM	2	142	144	141
DHT+XVSM	10	148	143	145
DHT+XVSM	60	169	149	156
DHT+XVSM	120	153	174	152
DHT+XVSM	180	157	164	150
DHT+XVSM	210	155	181	168

Table 3: Durations [ms] for the retrieval of a message out of 10, 100 and 1000 entries

Table 3 shows the benchmark results. In case of a plain-DHT solution, the measured overall retrieval-time consists of a lookup for the list of keys (stored under the well-known key), and additionally of 10, 100, and 1000 lookups for each of the keys stored in the value of KW. The presented numbers present the worst case. Consequently, in total a number of 11, 101, and 1001 lookups had to be performed, where a single lookup takes about 25ms. This means that in the best case (i.e. the first entry already matches the template) only two operations are performed, lasting about 50ms. As it can be seen, the increase in the path length with the number N of peers (which is proportional to O(log N)) does contribute little to the lookup time in the plain DHT case. It is the request and response processing that accounts for most of the 25ms needed by each

lookup. In this setting the caching capabilities of the DHT implementation were disabled (to avoid returning invalid values due to the high number of information updates).

With respect to the worst case scenario, the measured times for the integrated architecture (DHT+XVSM) are – except for a network of size 2 - significantly lower and almost constant in comparison to the plain-DHT solution. The time needed for the DHT only solution strongly depends on the number of entries for the region. The reason that the integrated approach performs better, is that there are only two operations to execute: a first lookup operation to retrieve the container reference, and a second direct request using an improved Linda-coordinator [115] to search for the proper message.

With respect to the best case scenario, the DHT only solution performs better. However, its advantage of being faster than the integrated solution lasts up to 5th – 7th message (143ms/25ms – 181ms/25ms). If the 5th – 7th retrieved message does not match the given template, the integrated solution is more efficient. Since it is given that more than 5-7 messages have to be stored for a region, it can be concluded that the integrated solution is more efficient than the plain one.

Furthermore, given the connectivity time window, where vehicle and RSU can exchange information, the integrated approach never violates the timing restriction since is always capable of retrieving information within that connectivity window of 2-3 secs. This means that the proposed approach is more stable. In contrast, the plain DHT solution strongly depends on the "right ordering" of the keys implying the probability of violating the 2-3 seconds of transmission time.

Efficiency of Information Dissemination

In the following we look into the delay constraints and throughput capabilities of the integrated solution. The implemented system was deployed on eight physical nodes simulating 200 RSU peers. The project partner, the Austrian highway operator ASFINAG, develops in the EU project COOPERS an infrastructure-to-vehicle communication solution in which special geo-located traffic and safety messages (coded in a markup format called tpegML – the XML implementation of Transport Protocol Experts Group Specification) are distributed to RSUs situated in Tyrol. As the nodes in our testing environment have identifiers in the modified hash space corresponding to coordinates on the highways A12 and A13 in Tyrol, the TPEG messages created by the Asfinag Traffic Control Center can be injected and processed on our system as well.

The messages are generated in real time by the traffic control center and stored in xml files. In order to generate processing load, these messages have been injected at once in the system via several peers. After being parsed, and transformed into an internal XVSM entry format, the messages were time stamped. After the dissemination to their final destination the messages were time stamped again, and the difference measured.

Figure 53 shows 1000 messages (i.e. the black dots in the diagram) entering the system via the eight nodes, i.e. 125 messages per node. The x-axis refers to the time when the message was sent while the y-axis shows the time the message lasted in the system before it was successfully stored in the destination container. This means that the last message was stored in the system after 31895ms whereas the dissemination of the last message took 72ms.

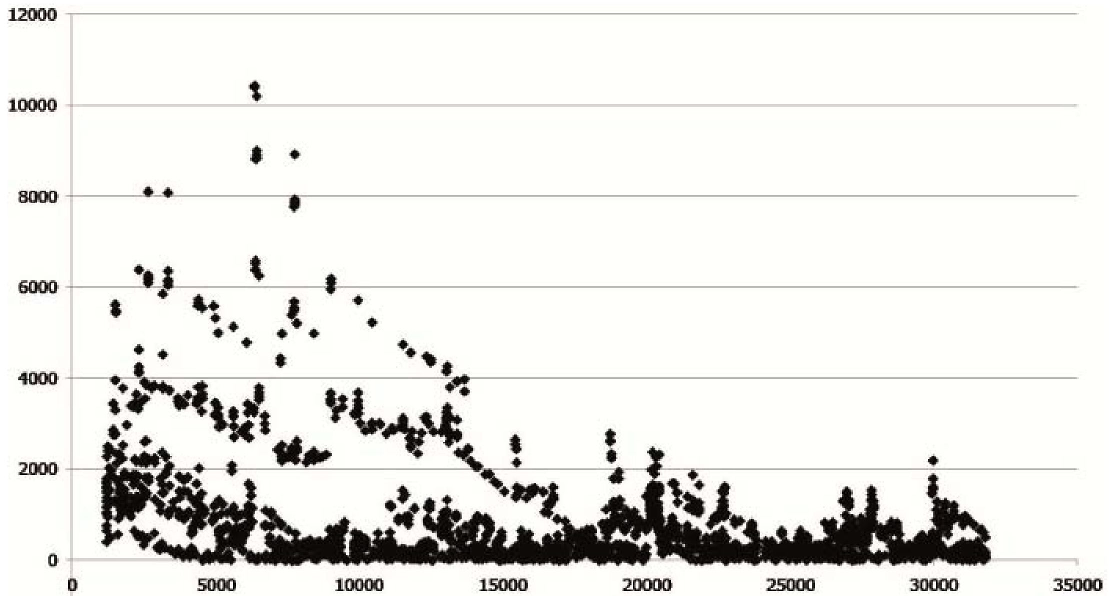

Figure 53: Time spent in the system versus message entrance time in ms [19]

The "patterns" in the figure are an effect of aggregation between the delays on the eight different RSUs: some RSUs are slower or have a larger processing load, others are much faster. However the bottleneck is finding the right peer in the overlay network for the message. It can be observed that at the beginning there is the effect of peaks because dissemination queues are getting bigger. The delay is caused by our implementation of the hash-function that uses x,y geo-coordinates to disseminate the messages. Delay can be minimized by means of improving the algorithm in the hash-function.

Figure 54 shows that between 70 and 100 messages per second can be processed by the system. However, the used DHT function is heavily used and limits the system's capabilities since requires 180ms on average per message to find the destination node for the message.

It can be discussed that the proposed integrated solution is effective in the sense that it is capable of disseminating all messages to the peer in the overlay network. The proposed solution was defined as efficiency if a publisher of information is capable of disseminating more information into the system than a vehicle driver can retrieve via an RSU within its transmission time. The transmission time is between 2000-3000 ms with a transmission capability of ca. 300KB/sec allowing the exchange of small and a few messages only [238]. The size of message to be transmitted varies between 5 and 10KB.

This means that between 60 and 180 messages can be transmitted to the vehicle driver. Figure 54 shows that the proposed solution is capable of disseminating 70 and 100 messages per sec. This implies that the architecture is not efficient enough in the context of disseminating information. However, the weak point is the hash function that needs 10 times more processing time as a hash function without geo-coordinates. If we succeed in improving the speed of our function comparable with ordinary ones, than the number of message the system can disseminate rises to 500 to 1000 messages. At this point the system would operate efficiently. However, this is appending future work.

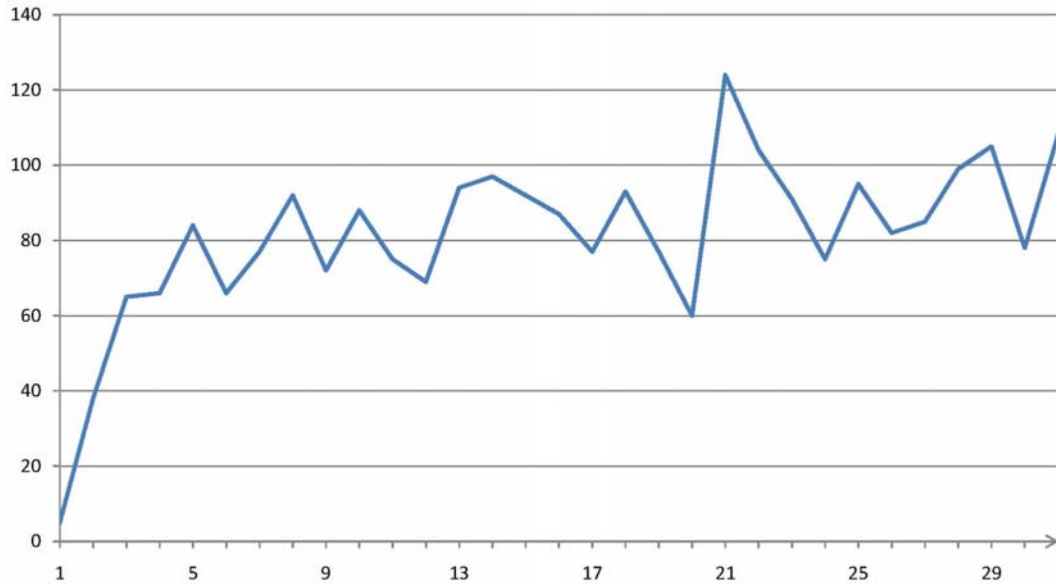

Figure 54: Total message throughput [19]

6.1.1.2. Agility of the Architecture

So far it has been described that the concepts of XVSM flexibly facilitate the integration of DHT concepts into an XVSM based software architecture at the communication and distribution category. As described in section 5.1.4.2 the proposed architecture makes use of extension profiles to replicate messages between different containers.

However, while a distributed application using Pastry is completely dependent on its built in replication features, the XVSM architecture enables any kind of replication strategies in extension profiles.

An extension profile may refer to a strategy where XVSM container replicas and DHT replicas are completely independent (i.e. decoupled) from each other. However, an XVSM replica manager has to be implemented to achieve this, as the replica mechanisms provided by the DHT middleware are not used anymore. In the case of a failing node during a write operation on a container, further actions don't have to be taken similar to the proposed solution. The difference is that in the decoupled solution

the XVSM replica manager and the DHT middleware will both have its own logic to determine when and where to create new replicas. Creating new replicas for the XVSM containers is the responsibility of the XVSM replica manager and not the responsibility of the DHT middleware. The separation allows the usage of any DHT technology and allows the deployment of any replication strategy. Furthermore, the network traffic is decreased in comparison to the proposed solution, because DHT and XVSM replicas will usually not be on the same computer. So, only one of the replicas will have to be recreated, if a computer fails. The disadvantage is that the implementation effort is increased in comparison to the proposed solution, because an own replication protocol has to be developed

Another approach is the mixed solution consisting of a list of container references for the replicas, like the coupled and proposed solutions. This list is not updated by any replica manager when a computer fails. Then there is an additional list of references to the nodes, where DHT replicas have been created by the DHT middleware. If there is a publish operation, lookup operation or an operation accessing the container, which discovers that a container replica is not accessible anymore, a new replica on the site is, which stores the second list. The advantage is the decoupling between container and DHT, but the middleware still has to provide the possibility to execute code as soon as DHT replicas are created or deleted. The disadvantage is that the replication mechanism is triggered by the lookup/publish methods. This could be too late, data could be lost.

6.1.1.3. Efficiency of Continuous Coordination Support

This section evaluates the effectiveness and efficiency of the integrated solution in the context of disrupted communication between publishers and subscribes. In the scenario the vehicle driver is the subscriber being disconnected from the system most of the time. The evaluation tests the effectiveness of implementing additional functionalities being supported by aspects in the XVSM architecture and the efficiency of retrieving messages the subscriber subscribed for. The architecture is efficient if the number of relevant transmitted messages is higher than the number of messages transmitted via traditional publish/subscribe systems.

For the evaluation of this issue we have implemented a scenario in which an application was used to monitor the amount of vehicles passing an RSU for within the last 60 seconds and calculate the average speed of all vehicles. Every passing vehicle had to send its current speed to the RSU which stored this information and provided it to vehicles which were interested in it.

We compared the approach with an implementation using durable message queues from the Java Message Service (JMS) [150]. We decided to use JMS because it is an acknowledged standard developed by Sun Microsystems and implemented by several well-known commercial and non-commercial providers. The JMS API provides two

messaging models: queuing and publish/subscribe. The queuing model is a one-way point-to-point communication between a sender and a receiver. The sender writes messages into a specific queue, wherefrom the receiver consumes the messages in a first-in-first-out manner. The publish/subscribe model is described in section 5.1.3.3.

The simulation consisted of a single client application, representing the vehicle, and a single peer, representing the RSU. In the first test, the peer was running a JMS server and in the second one the XVSM without DHT, since no distribution is needed. We used the traffic statistics of the Austrian highway "A23", which is the most frequently used highway in Austria. In 2008, 153100 vehicles used the highway daily[18] in average. Assuming that the amount of cars is consistent over the day, approximately 1.56 vehicles use the highway every second. Additionally, in case RSUs are placed every four kilometers, it means that a vehicle passes the next RSU approximately after 3 minutes because the speed is 80 kilometers per hour. Every time a vehicle drives along a RSU, it reports its current speed and fetches the statistics from the RSU. This means that after 3 minutes an RSU has processed messages coming from approximately 280 vehicles, resulting in 280 events to be processed by the publish/subscribe system.

The amount of data to be transmitted is limited to 300KB per second according to the DSRC protocol (section 3.3.1). The size of the messages is between 5KB to 10KB (depending on the content) and the connection window of the roadside unit is only 2-3 seconds

In the JMS implementation we used a durable message queue to store the messages. Every vehicle sends a message containing its current speed to the queue when it is passing by the RSU. Additionally, it retrieves all the messages which have been published by the other vehicles. Since the JMS standard does not define any way to pre-/post-process messages, the vehicles have to read all messages from the queue, count them to get the amount of cars which passed the RSU and calculate their average speed using the contents of the messages. Due to the technical restrictions of the transmission protocol, it may occur that it is physically impossible to transmit all messages from the queue to the vehicle.

In the XVSM implementation we used an aspect which calculates the average speed. Every vehicle passing the RSU sends its current speed to the RSU. Instead of storing all the messages in the container and providing it for interested parties, an aspect processes the messages and writes the result of the calculation into the container. When a vehicle is interested in the traffic statistics it will receive only one message which contains the aggregated result. Therefore the amount of messages is decreased (Figure 55).

---

[18] http://de.wikipedia.org/wiki/Autobahn_S%C3%BCdosttangente_Wien

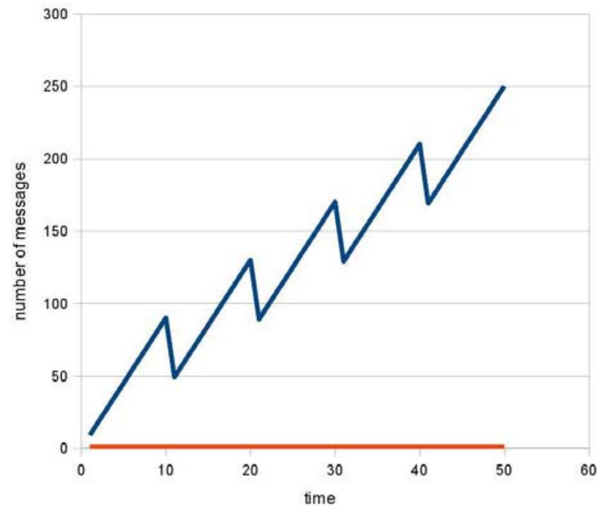

Figure 55: The development of the size of a message queue in case data cannot be retrieved sufficiently by vehicles within the connection window

The blue line in Figure 55 shows the number of messages when JMS is used depicting the increasing amount of data within a message queue in case messages cannot be retrieved by the vehicles within the connection time windows. The orange (horizontal) line shows the amount of messages in the XVSM based implementation. In case vehicles are in transmission range, they try to fetch as many messages as possible. The only limit is the connection time window. Therefore, every time the blue line falls, messages have been retrieved. However, since - due to low transmission capabilities - not all of the messages can be transmitted the amount of messages increases continuously. In contrast, the number of messages in the XVSM based implementation is always one because it stores an aggregated message only.

The results show that the XVSM solution is more efficient as the JMS solution. Aspects play an important role in this context, because they represent logic which helps the subscribers to retrieve already preprocessed messages. This enables that the subscriber reconnecting to the system is already up-to-date without the need to retrieve and process messages locally, as in the JM solution.

In addition to the simulations described above we implemented several performance tests using XVSM combined with DHT implementation for the case that a subscriber has to retrieve several entries instead of a single, aggregated one. Table 4 depicts the time it takes to retrieve 10, 100 and 1000 entries from a container in a network with various numbers of peers. As shown in the table the time to retrieve the entry is independent of the number of peers in the DHT. This shows that the overhead to lookup the container in the DHT and to execute a query in the container does not have negative influences on the performance of the system.

Configuration	Peers	10 Entries	100 Entries	1000 Entries
DHT+SBC	2	142	208	1378
DHT+SBC	10	148	258	1002
DHT+SBC	60	169	256	1115
DHT+SBC	120	155	256	1184
DHT+SBC	180	155	231	1086
DHT+SBC	210	155	231	1086

Table 4: Durations [ms] for the retrieval of 10, 100 and 1000 entries [123]

6.1.1.4. Complexity Analyzes

The aspect of complexity is discussed by comparing the XVSM solution with an alternative plain-DHT implementation with respect to retrieval and dissemination of messages, and the JMS solution with respect to publish/subscribe communication in disrupted environments.

Information retrieval and dissemination

Figure 52 shows a diagram with the sequence of operations regarding information retrieval. As it can be seen, the number of processing steps the application component has to perform is in case of the plain-DHT solution equal or greater than in the integrated solution. It is equal if there is only a single message for the region, or if already the first message matches a given search template. In any other case it is greater than in the integrated solution. Already based on this distinction (retrieve all messages vs. retrieve a single message) the complexity of the implementation of the application component rises because it has to implement both methods. In case of more complex queries (e.g., retrieve all messages matching two distinct queries) the complexity of the application component rises as well, since these features have to be implemented there. The integrated solution needs two operations only: First, the application component has to look up the container, and then in a second step execute the query. While in a plain-DHT solution the number of processing steps is unknown, from the perspective of the application component the complexity of a query is consistent, since it is reflected in the coordinator.

With respect to information dissemination the complexity of the application component in the plain-DHT solution is higher by a single operation. In the plain-DHT solution, the application component has to look up the well-known key KW, and then perform two put operations: one is the new value of KW that has to be written, and the other refers to the new message. In case of the integrated solution two operations have to be executed.

First, the application component looks up the container, and in a second step performs a *write* operation on the retrieved container.

Publish/subscribe communication in disrupted environments

The limitation of current publish/subscribe systems with respect to the given application domain is that they do not support mechanisms allowing the aggregation of messages already in the system. In contrast to the integrated solution in which the separation of concern is clean, this implies that aggregation strategies have to be managed and executed by RSUs. RSU would act as intermediate subscribers and aggregate information according to the subscriber's policy. However, the main responsibility of an RSU is to transmit messages to and receive messages from passing by vehicles. Implementing this requirement would a priori increase its complexity. Nevertheless, its complexity has to rise even more, since vehicles are moving connecting themselves to other RSUs when they could establish a connection. This implies that the RSU is burdened with tasks it has not been designed for. It has to search for the RSU that stores messages already aggregated for the subscriber, retrieve them, and send them to the vehicle. Furthermore, the aggregation logic has to be moved from the previous RSU to the current one.

6.1.2. SWIS Application Scenario

The aim of the SWIS project is to develop an "information sharing network" for the Air Traffic Management domain. Due to the safety-critical domain a manual configuration of the architecture is not allowed. The proposed MDSC approach creates of capability and requirement models of services and heterogeneous network infrastructures, a configuration that is used by the architecture to adapt to changing requirements.

The following evaluation contributes to the thesis' claims in section 3.1.1 and 3.1.2 regarding the support for flexible software architectures (C2.1), robustness against changing requirements (C2.2), and reduction of complexity of application components (C2.3).

Flexibility means the ability to change or react when necessary while robustness means the absence of a need to change or to react. The section investigates the claims by comparing and evaluating the current SWIS architecture with an exemplary architecture using message-oriented middleware (MOM). The proposed architecture is more flexible if it can cope with changes in the sense that the number of changes to the architecture is fewer than in case of MOM. The proposed architecture is more robust if it does not require the integration of other architectural styles or the adaptation of application components. The proposed architecture is efficient if complexity is reduced, i.e. the number of architectural concepts is kept minimal.

For evaluation purposes we defined two scenarios which show the advantages of the proposed architecture in comparison to message passing between the components. A message-oriented middleware, like Java Message Service, is an appropriate technology for comparison since messages are passed from one architecture component to the next where they are processed individually.

6.1.2.1. Robustness of Architecture

The defined scenario starts with a basic setting that is subject to changes which were derived from section 5.2.1. The changes are processed and mapped by MDSC onto a configuration that is to be used by each SWIS node.

Initial Situation

The initial situation refers to the configuration between the Routing Component and the Transformation Component. In this setting the configuration of the Transformation Component refers to a transformation instruction specifying that the structure of the message has to be changed. Figure 56 shows the different concepts for communication regarding the initial scenario. While on the right hand side the XVSM concept (SWIS-XVSM) is shown (section 5.2.3), the left hand side presents the implementation using queues (SWIS-Queue).

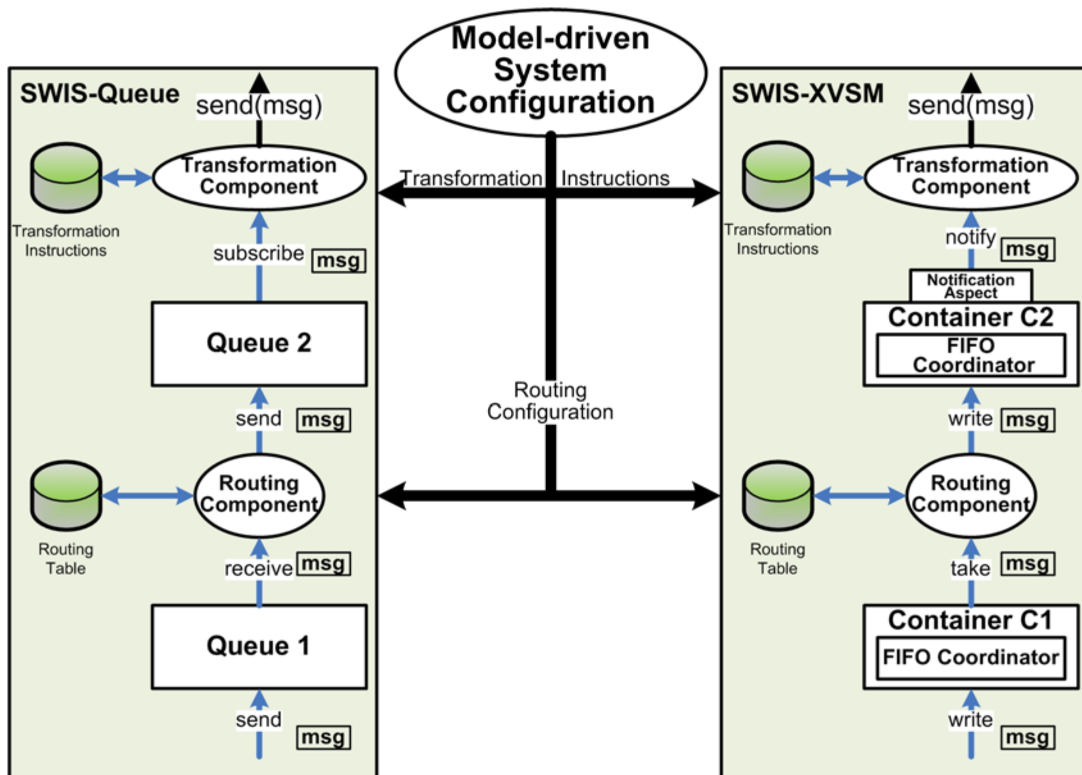

Figure 56: Communication between components using queues and containers with a simple transformation instruction

In the SWIS-XVSM implementation the processing of messages are as the following. Incoming messages are placed into container C1. After the Routing Component has decided where the message has to be forwarded to, it passes the message to the Transformation Component by writing it into its container C2. The Transformation Component registered a consuming notification (section 4.2.3.2) on the container pushing every message to the Transformation Component and deleting every written message from the container.

In the SWIS-Queue implementation incoming messages are placed into Queue 1. After the Routing Component has decided where the message has to be forwarded to, it passes the message to the Transformation Component by writing it into its queue, Queue 2. The Transformation Component registered a subscription on the queue pushing every message to the Transformation Component.

The difference between the two implementations is not the semantics of the connector between the two components, but its realization. While on the right side containers hosting a FIFO Coordinator represent the connector, on the left side queues do.

Changing Requirement

In the new configuration the MDSC has derived that instead of a simple transformation of the message structure messages of several different types have to be merged together before passing it to the service. Figure 57 shows the updated architectures of the SWIS nodes which is capable of satisfying the given requirement.

On the right hand side the SWIS-XVSM architecture does not change but the Transformation Component specifies a different subscription to a consuming notification which is managed by the notification aspect. The new subscription induces the notification aspect to group messages according to specific message types and to push them combined to the Transformation Component. The Transformation Component receives a number of messages each with different types which in turn are merged according to the transformation instruction.

In the SWIS-Queue architecture several changes have to be performed due to limitations of queues. Although, several different subscriptions may be registered at a queue, messages matching those subscriptions are immediately pushed to the registered application components. According to the transformation instruction the Transformation Component has to subscribe for several different message types which have to be merged. The problem is that it is not specified when messages of each type arrive. This implies that several messages of the same type may arrive one by one, and then the other message types. However, the Transformation Component changed the structure of messages so far. Therefore, the new requirement forces the SWIS-Queue architecture and the Transformation Component to change as well. The SWIS-Queue architecture has to be extended by a storage component temporarily storing all incoming messages and by an additional logic (Storage Logic) that stores messages of each type and notifies

the Transformation Component if at least one message of each type is available for merging.

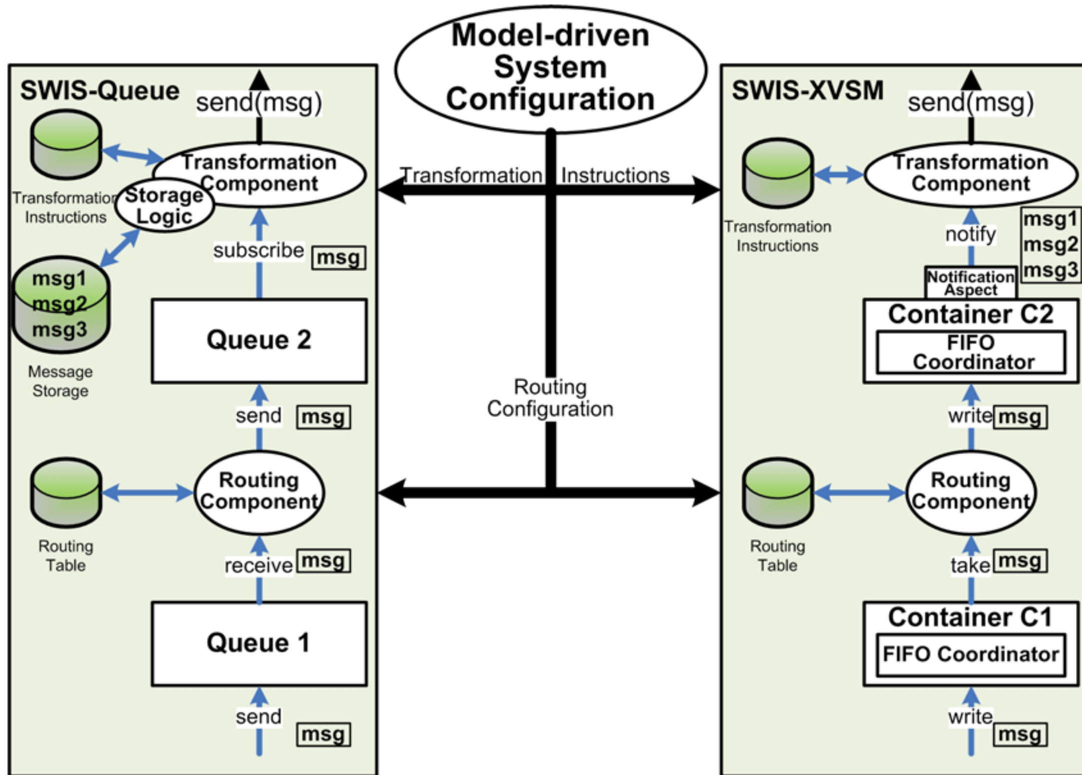

Figure 57: Communication between components using queues and containers with an aggregating transformation instruction

The introduction of a new requirement has been processed in different ways with different outcomes. While the SWIS-Queue architecture had to be extended by a new architectural style (data-centric architectural style) and the Transformation Component by logic managing that style, the SWIS-Queue architecture remained robust to the new change and the Transformation Component was not affected either.

6.1.2.2. Flexibility of Architecture

The defined scenario starts with the setting of Figure 57 that is used as subject for further changes derived from section 5.2.1. Changes in the configuration were triggered due to the installation of shadow nodes (section 5.2.3.4) implying a change in the capability models of the infrastructure. The changes are processed and mapped once again by MDSC onto a configuration that is to be used by each SWIS node.

Figure 58 shows the architecture of both concepts. In the SWIS-XVSM implementation the architecture itself has not been changed. The SWIS-XVSM architecture is flexible in this context, because a pre-aspect (S-Aspect) responsible for synchronization between

containers hosted on the main node and the shadow nodes has been added to the container without affecting any components. In contrast, the SWIS-Queue architecture had to be changed by introducing additional logic (S-logic) placed between queues and components responsible to synchronize queues between the main node and the shadow nodes.

The implementation of S-logic could have been avoided if MDSC derived a proper configuration for clustering, since most vendors of the JMS standard support clustering of queues. However, this would make MDSC a platform specific approach with high complexity rather than a solution for a platform coping with heterogeneous technologies. Furthermore, MDSC derives extension modules specifying the semantics of clustering in the Cluster Module Specification. This means that the architecture also has to be capable of integrating non-vendor specific components for replication of messages and keeping the nodes synchronized.

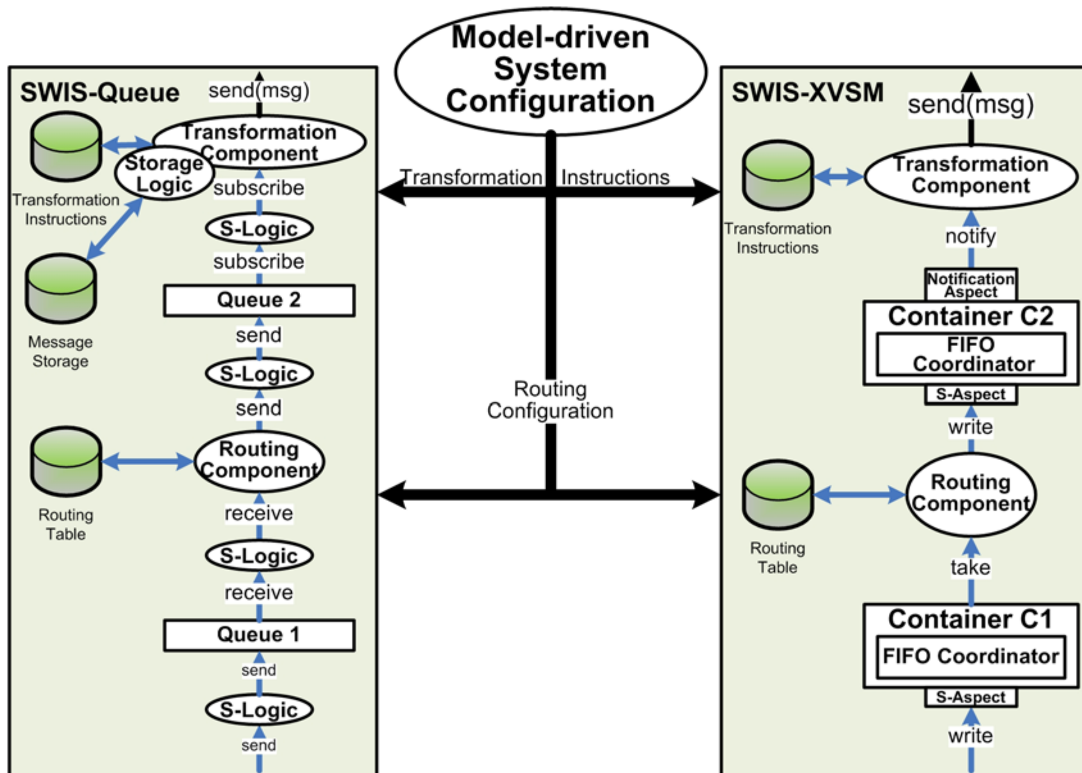

Figure 58: Communication between components using queues and containers in case of Shadow Nodes

6.1.3. SAW Application Scenario

The SAW project investigates the capabilities of the SBC architectural style with respect to the reintegration of failed application components into the coordination process within a scenario from the production automation domain.

The following evaluation contributes to the thesis' claims in section 3.1.1 and 3.1.2 regarding continuous coordination support for recovered application components (C1.3), and reduction of complexity of application components (C1.4).

The claims are investigated by discussing complexity factors and the effectiveness and efficiency of the SBC architectural style with respect to continuous coordination support by comparing SBC with traditional solutions. The proposed approach is effective if it is capable of providing the same functionality as traditional solutions, while it is efficient if complexity is reduced, i.e. the number of architectural concepts remains equal and the number of processing steps decreases in comparison to traditional solutions and architectures. In the second aspect the proposed implementation is efficient, if it is capable of integrating application components earlier into the coordination process than messaging systems.

In order to answer these aspects, the scenarios from the production automation domain were implemented and evaluated by means of the Manufacturing Agent Simulation Tool (MAST) [224]. MAST is a suitable tool providing agent-based simulation support for an empirical study of recovering agents developed by Rockwell Automation.

6.1.3.1. Efficiency of Continuous Coordination Support

In the following sections we measure the time needed to initialize the proposed concept and the effort needed and time taken to fully recover in case of single and multiple agent failures. The evaluation analyses and compares the processing and effort needed to recover an agent to a state where it supports "optimal" production. The evaluation takes into account the traditional message-based approach and the MozartSpaces implementation of XVSM for comparison. The benchmark results presented are averages of ten test runs conducted on a Core2Duo T9400 2 x 2.5 Ghz, 4 GB RAM. MAST is a single client simulation tool where simulated agents are not distributed over several peers but run in a single Java VM.

Initialization

Evaluation of initialization time discusses the time needed for a single agent (Figure 59) and for the entire production system with several agents (Figure 60).

When the system is started the first time all existing CAs exchange information about incoming and outgoing CBAs. An exchanged message contains information about e.g., costs like the time needed for a product to move to the next intersection. Based on that information each CA builds up its own routing table that contains the shortest path to each existing MA. This ensures that incoming PA can be routed quickly and that the overall production system is working "optimally" with respect to the defined objectives.

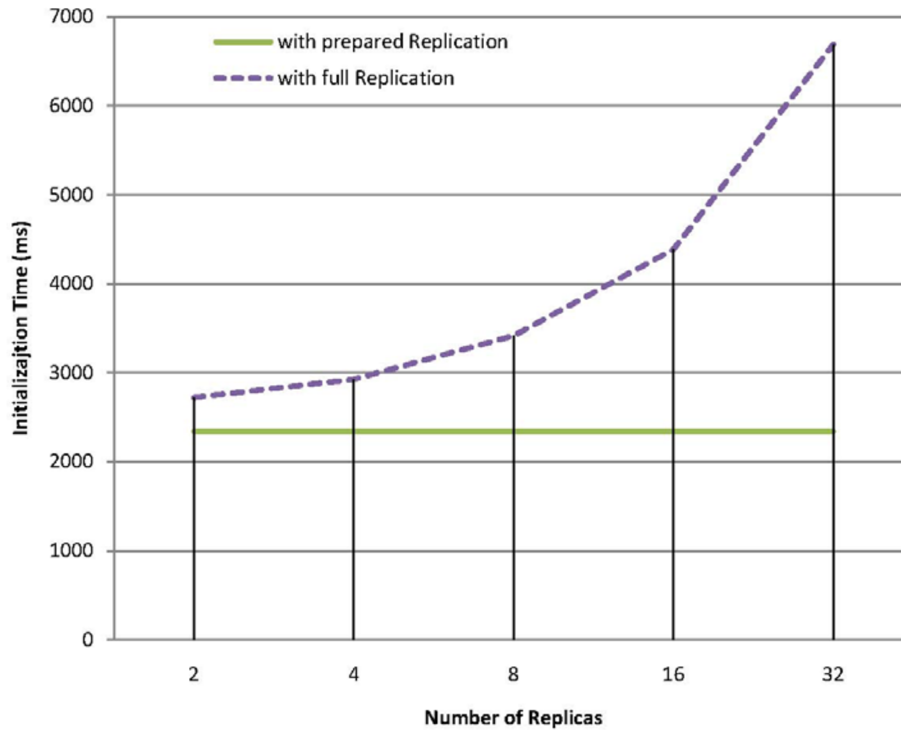

Figure 59: Initialization time for a single Crossing Agent

However, the processing steps of creating the first routing table have to be performed anyway independent of the used replication strategy. The time needed to exchange all information is minimal and takes about 15ms per agent to have its view on its routing table. In case of using MozartSpaces additional time is needed to create and configure containers, which takes about 83ms per agent.

Since MozartSpaces facilitates the replication of containers the time needed to set up replication has to be added to the init-phase of the agent as well. Figure 59 shows the time needed to create n replicas of a container (green non-dotted). For keeping replicated containers consistent we used the JGroups[19] implementation. Therefore, beside the creation of replicated containers also the time needed to install and start aspects containing the logic of JGroups had to be measured (purple dotted line).

It can be seen that the time needed to prepare replication and to fully configure an agent adds significant amount of time to the init-phase of the agent, starting at additional 493msec in case of two agents and almost exponentially increasing with the number of agents. In case of 32 agents the additional time is about 4500msec. The additional time comes from loading JGroup specific multicast libraries and configuring JGroup with multicast parameters. Since the simulation runs on a single peer delay is introduced due to port conflicts. The more replicas have to be maintained the more conflicts rise delaying the init-phase of the agent. In a distributed environment the time needed for

[19] http://www.jgroups.org

this step would be constant as well, since setting up replicas can be performed concurrently.

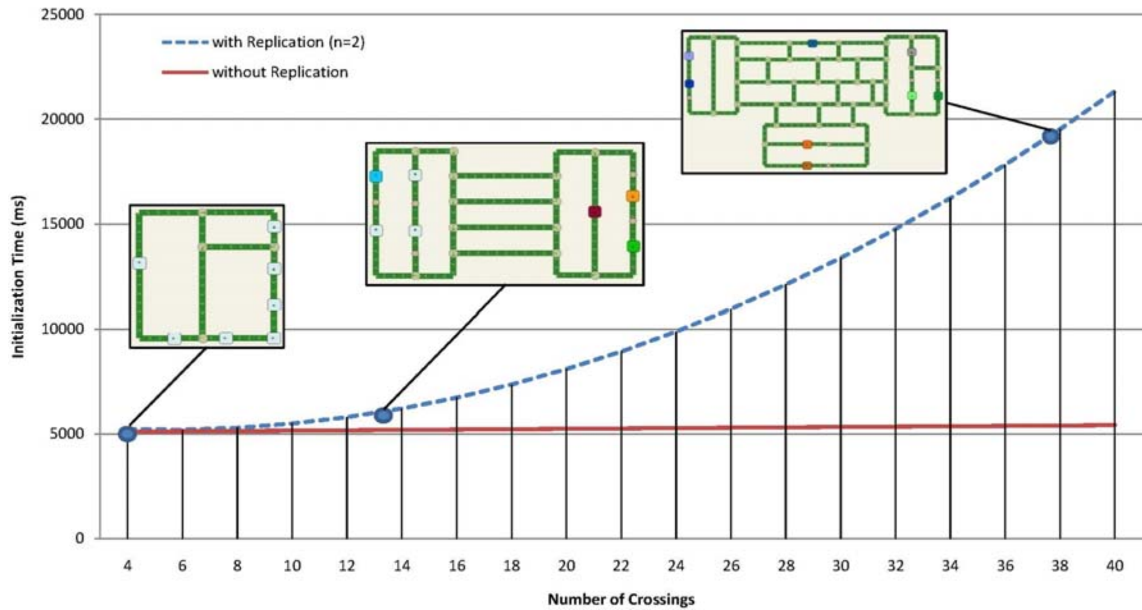

Figure 60: Initialization time for multiple crossing agent

Figure 60 shows the time needed to configure the entire production system. The red non-dotted line refers to the time needed to setup the system without replication. As it can be seen the line is not constant. This is the case, as the time needed depends on the number of running CAs and the complexity of the layout. The more CAs exist and the more complex the layout is, the longer it takes. The reason is that more routing information has to be exchanged between the CAs to setup a local view of the routing table. The blue dotted line shows the time needed to deploy a replication strategy that keeps two containers consistent. As it can be seen the time needed for replication significantly prolongs the init-phase. In case of 40 agents it takes more than 4 times longer to get the system running.

Recovery Effort & Time

Concerning recovery it has to be distinguished between a single agent crash and multiple agent crashes. Table 5 shows the time needed to recover a single agent and the steps to be processed. Results show the evaluation in a local environment, whereas message delay was considered constant and equal for both approaches since information has to travel the same physical distance.

Recovery by means of message exchange with other neighboring agents is started by requesting (step 1) the routing tables of agents placed on the other end of all outgoing conveyor belts of the recovered agent. In case the agent has several outgoing conveyors, the agent has to place the same amount of requests resulting in the same amount of responses (step 2). However, it cannot be determined when responses arrive, implying

the possibility of updating the local routing table (step 3) and informing neighboring agents of incoming conveyors about new conditions (step 4) several times. A timeout occurs if the requested agent is not reachable (problem of multiple agent crashes). Therefore, the time needed to recover a single agent takes at least 10ms. The reason is that updating its routing table depends on the time when the last message comes in (i.e. this also includes the probability of timeouts due to multiple crashed agents). If any other messages arrive then it means that the agent has not been in the state it would have been, if it had not crashed, and the production system is not processing in an "optimal" way as required.

Process	Message-based	XVSM
Step 1	requesting routing tables of outgoing crossing neighbors	lookup of replica
Step2	receiving routing tables (or timeout)	restoration of local container
Step3	building of new routing table (using received ones including conveyor belt costs)	notification of ingoing crossing neighbors
Step4	notification of ingoing crossing neighbours	
Time	at least 10ms or timeout	at least 10ms

Table 5: Comparison of crossing recovery; message based vs. replicated Space Containers

Recovery by means of the MozartSpaces approach is done in three steps. In step 1 a lookup is performed in order to find a replica. In step 2 the local routing table is restored. This means that the entries stored in the replica are copied to the local container. Finally, in step 3 the agent informs neighboring agents of incoming conveyors about the new condition. Although, we have described three steps, the agent would only need to look up and access the container. The migrating of content from the replica to the local container is done transparently. The time to execute the process takes at least 10ms. It is comparable with the previous approach, but it can be assured that the agent is already up-to-date without bothering other agents with additional updates.

Simulation of the failure and recovery of multiple crossings at the same time indicates the performance potential of the presented solution in section 5.3.3. Consider the subsequent failures of crossing CF1, CF2 and CF3 (Figure 50), followed by their recovery in the same sequence (detailed in). The more subsequent recoveries take place the more efficient the proposed approach will be in comparison to a message based approach. In case of a catastrophic failure (i.e. power cut to DHT lookup region 1) failure detection is only given between functional components (the ones in DHT lookup region 2) and their failed neighbors, leading to the same steps of recovery as described previously and the same result in the end.

	Message-based	Replicated containers
CF1	steps 1 to 4 of (wait till timeout to discover failure of CF2)	steps 1 to 4 of
CF2	steps 1 to 4 of (wait till timeout to discover failure of CF3, step 4 leads to computing of steps 3 and 4 at CF1)	steps 1 to 4 of (step 4 leads to a single write-operation in the container of CF1)
CF3	steps 1 to 4 of (wait till timeout to discover failure of CF3, step 4 leads to computing of steps 3 and 4 at CF2 and subsequently to steps 3 and 4 at CF1)	steps 1 to 4 of (step 4 leads to a single write-operation in the containers of CF2 and CF1)

Table 6: Comparison of multiple crossing recovery; message based vs. replicated containers

6.1.3.2. Efficiency of Coordination

One of the major challenges in production automation is the need to become more flexible in order to support the fast and efficient reaction to changing business and market needs. However, the overall behavior of the many elements in a production automation system with distributed control can get hard to predict as these heterogeneous elements may interact in complex ways (e.g., timing of redundant fault-tolerant transport system and machines) [137].

An approach towards fast reactions may be the prioritization [114, 187] of pallets. Some special parts of the product with higher priority have to be favored by the agents rather than pallets with lower priority. This approach may help to a) produce a small number of products quickly, or b) to phase out products as soon as possible in order to free resources for new products to be assembled. Therefore, the aspect of priority has to be considered between all neighboring crossing agents (CAs) and all conveyor belt agents (CBAs) connecting them. In the described scenario a CA has to check first, whether there is a pallet with high priority on one of the transporting conveyor belts. If this is the case that particular CBA may speed up its transportation speed as well as the CA may force the other conveyor belts to stop. This may happen by e.g. either not handling any pallets coming from them and so forcing those CBAs to stop, or by requesting the other CBAs to halt. So, the high priority pallet is being routed earlier than the other pallets, and it has overtaken other pallets which may have occupied machines needed by the prioritized pallet based on its production tree.

Simplified, the scenario can be summarized as the following: entries have to be ordered by means of the sequence of writing and grouped according to the priority of the entry written. Then, the task is to remove the entry first written from the non-empty group with the highest priority. Additionally, a conveyor belt has only a limited amount of

space available depending on the length of the conveyor. In the following the proposed architecture is compared with a JMS messaging middleware and Linda tuple space.

Java Message Service

For communication between agents in the production automation system, JMS [150] queues are appropriate. With respect to the described statement, Figure 61 depicts on the left side how queues would realize the coordination problem with three different priority categories whereas 1 is the highest priority. The right side of the figure shows the realization with a container containing a PRIO-FIFO Coordinator. The PRIO-FIFO Coordinator stores messages in a FIFO order grouped according to their priority. Additionally, both diagrams show the sequence to write an entry and to take the next entry with the highest priority from the FIFO perspective.

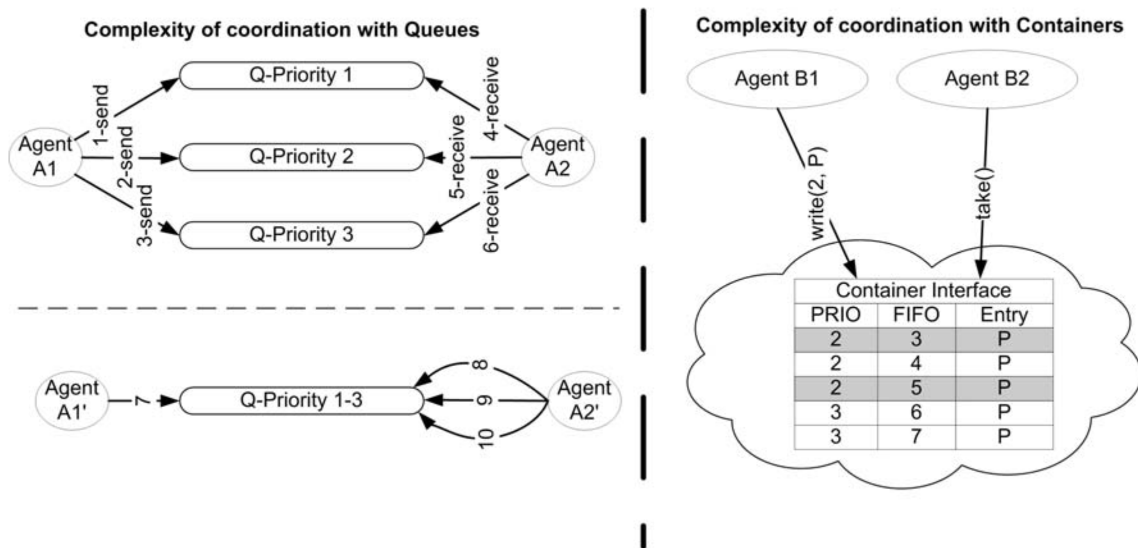

Figure 61: Comparing the complexity of prioritized queues with the container concept (P..entry)

In case of queues there are two possible implementations. In the first one there is one queue for each priority. In the second solution a single queue hosts all messages (i.e. entries) whereas parameters in the message header define its priority for which so called selectors allow querying.

In the first solution, when an agent (Agent A1) wants to place an entry into a queue it looks up its priority. Based on the entry's priority the *send* operation (operations 1, 2, or 3) of the proper queue is executed. This implies that the application component has to manage three different queue connections. However, before placing the entry into the queue the agent has to retrieve its size. If the number of stored messages is greater than the maximum of permitted ones, then the sender has to look for alternative routing paths. On the receiver side, the agent (Agent A2) has two option of how to receive an entry (operations 4, 5, or 6). Either it polls queues starting with the queue with the highest priority, or it is notified by JMS in case an entry has been written into one of the queues. If it polls, then the agent accesses the queue with the highest priority (Q-Priority

1, operation 4) first. If it is empty then it accesses the queue with the second highest priority (operation 5), and so on. Once a queue has been found that is not empty it removes the entry from the queue and processes it. If the agent is notified then messages are pushed to the subscribed agents. However, in this case the concepts of a queue have to be changed from *QueueSession* and *QueueReceiver* to e.g., *TopicSubscriber* and *MessageConsumer* triggering an update of the agent's implementation logic. The difference between the two approaches is mainly concerned with the question of who controls an agent. If the agent is notified then it has to process the pushed entry immediately. If the agent polls a queue it can act more autonomously since it can specify when to access a queue and according to which strategy (e.g., configuration of polling rate). In the second possible implementation (bottom part of the left side of), agents (Agent A1' and A2') access a single queue. The difference to the first implementation is the usage of selectors specifying the priority of entries to be accessed. This means that instead of three different connections to a queue, three different selectors have to be used appropriately.

In the proposed architecture (right side of Figure 61), the usage of the PRIO-FIFO Coordinator allows the software developer to specify the coordination policy transparent to the agents. A *write* operation needs a priority parameter and the entry. How entries are stored in the coordinator is up to the software developer and of no concern to the application component (coordination category). Since the coordination policy is represented in the coordinator the agent's *take* operation already reflects its semantics regarding priority restrictions. This means that the *take* operation does not need any parameters as the coordinator already knows that the entry with the highest possible priority has to be returned.

The migration from a *take* operation to a *notification* of written entries does not imply any change of concepts. The application component has to execute a *notify* operation where it specifies the callback method. As described in sections 4.2.2.1 and 4.2.3.2 aspects make sure that consuming notifications are pushed to the application component. In contrast to the three queues, aspects can also help sort notifications according to the concurrently written entries' priorities before delivering them to the application component.

Linda Tuple Space

Figure 62 depicts on the left side how the Linda tuple space approach would realize the coordination problem. The right side of the figure shows the realization with a container containing a PRIO-FIFO coordinator. Additionally, both diagrams show the sequence to write an entry and to take the next entry with the highest priority from the FIFO perspective.

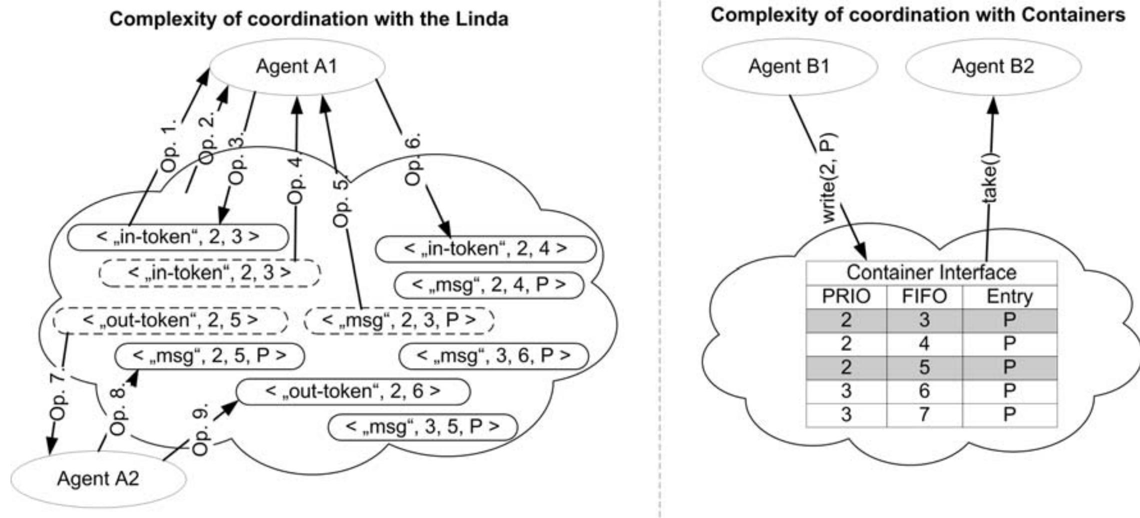

Figure 62: Comparing the complexity of a prioritized queue of the traditional Linda approach with the container concept (P..entry)

For the implementation of a queue in Linda two additional tuples have to be placed into the tuple space. One tuple that represents the first index (i.e. beginning) of the queue (in-token) and one that represents the last index (i.e. end) of the queue (out-token). Therefore, each tuple in the space has to follow a specific structure. Either it is an index tuple containing information about its index type (in-token or out-token), the priority of the queue representing, and the actual value of the index, or it is a message type consisting of its type (i.e. message) and its index in the queue. Whenever a tuple is placed into the queue the last index tuple has to be taken out, the new tuple and an updated index tuple (i.e. index is increased by one) written into the space. Whenever the first tuple needs to be read, the first index tuple has to be found, its index read, and according to this information the tuple retrieved. Whenever the first tuple needs to be taken out, the first index tuple has to be found, its index read, the message based on this index taken out the space, and an updated index tuple (i.e. index is increased by one) written into the space. If no message can be retrieved then it implies that the current queue is empty. Therefore, the process has to be repeated until a message has been found with a lower priority.

Listing 3 shows how to retrieve an entry based on Figure 62 as an example setting for stored entries in queues. It can be seen, that while the XVSM approach needs a single operation to write or to retrieve an entry from the space, the Linda tuple space approach requires at three operations: one to remove the index tuple, one to remove/write the message, and one to write back the index tuple. This is because the realization of a prioritized queue requires the agent taking over a part of the coordination problem.

Listing 3: Retrieving a FIFO sorted entry with Linda.

Nr.	Operation
1.	`//retrieve index of first message with highest priority 1` `index = in("in-token", 1, ?int)`
2.	`//retrieve message from index with highest priority 1` `message = inp("msg", 1, index, ?P )`
3.	`// write back retrieved index tuple` `out("in-token", 1, index)`
4.	`//retrieve index of first message with new priority 2` `index = in("in-token", 2, ?int)`
5.	`//retrieve message from index with new priority 2` `message = inp("msg", 2, index, ?P )`
6.	`//write back new index tuple of new priority 2` `out("in-token", 2, index+1)`

Measured times required to retrieve the next entry, with highest priority, from a prioritized queue is shown in Table 7. A benchmark has been set up, which compares the performance of a JavaSpaces (as a Linda tuple space implementation), and a PRIO-FIFO Coordinator. The benchmark demonstrates that a PRIO-FIFO Coordinator is able to retrieve entries faster than a coordinator with Linda pattern matching techniques.

Entries	Linda	PRIO-FIFO
10000	5,24	0,20
20000	15,15	0,20
30000	47,93	0,21
40000	58,66	0,20
50000	70,10	0,21

Table 7: Time in ms to retrieve a single entry using different coordinators [124]

In order to run the benchmark the container was first filled with a specific amount of entries (10000, 20000, 30000, 40000 and 50000 entries). After that a *take* operation was issued, and the time needed to get the entry measured. The results of the benchmarks clearly show that the PRIO-FIFO Coordinator is always the fastest. The results also show that the PRIO-FIFO Coordinator offers constant access time, thus perfectly representing the coordination requirements within a single operation call.

6.1.3.3. Complexity Analyzes

As shown in the previous chapter, the complexity of the agent implementation can be reduced to two single operations: look up and query of a container avoiding any implementation issues within agents contributing to recovery.

For this chapter we analyze an alternative solution, namely the usage of local-lightweight databases for the persistent storage of the routing table in connection with message-oriented middleware technologies for the communication between agents. In such a setting the complexity of agent implementation increases because the implementation has to integrate and handle several different concepts at the same time:

The agent has to manage implementation logic that establishes a connection to the **database** and performs specific queries, thus increasing the agent's complexity. If the used database technology does not support **replication** or consistency management then the agent has to implement such issues. Furthermore, the application component has to be extended by **implementation logic** knowing how to update routing tables, and when and where changes have to be propagated for further processing.

In comparison to the previously explained architecture, an agent has to implement logic dealing with databases, replication, and update implementation logic. This means that the number of concepts the software developer has to know is higher than in the proposed architecture.

In contrast, the XVSM architecture already functions as a data storage component since its coordination model integrates the data-driven coordination model. Replication logic (distribution category) and implementation logic for propagation of updates (organization category) however is added and managed transparent to the application logic.

6.2. *Studies*

In this section we evaluate the usability of the SBC architectural style and compare it with alternative architectural styles by means of exercises carried out in lab courses at the Vienna University of Technology. Based on the conducted study, the presented evaluation results are contribution to support the thesis' claims in section 3.1.1 and 3.1.2

regarding the subsumption of both control- and data-driven coordination models (C1.1), and reduction of complexity of application components (C1.4).

The intention was to perform an empirical evaluation by means of two scenarios The first scenario focuses on simple information transmission between two application components, while the second scenario comprises the realization of complex coordination requirements with several participating application components. In particular, the first scenario is similar to the presented SAW use case. However, the exercises did not define any restrictions allowing students to use a production system with a defined production layout e.g., to produce unlimited amount of products based on conveyor belts with unlimited capacity. The second scenario is an updated version of the game Scrabble. In contrast to the traditional rules of that game, in the lab assignment it had to be decided within a predefined amount of time which of the players had the longest word. For this player it was then allowed to place his/her letters on the board.

Beside MozartSpaces or XCoSpaces students had to use any kind of alternative communication framework, but the same technology setup had to be used in both scenarios. Beside MozartSpaces, the most preferred ones were CORSO, JavaSpaces, and RMI.

The evaluation considered 3 courses and the scenarios were implemented by 68 groups, each consisting of two students. The challenges of the two scenarios had to be realized by means of various communication or coordination frameworks, while gained experience, realized architecture design, and implementation effort had to be documented and presented in front of class at the end of each course. The mixture of students reached to average ones with respect to the time needed to implement a scenario.

Table 8 summarizes the student's results concerning usability, while Table 9 presents the effort needed to realize the two scenarios both in time and lines-of-code (LOC). In Table 9 the effort is specified in percentages where MozartSpaces functions as basis for comparison. For instance, if a framework requires 50% effort, than it means that it needs half the time of implementing it than with MozartSpaces. 200% LOCs e.g,. mean than it means that the student had to write twice as many lines of code as with MozartSpaces. The definition of LOC refers to only those implementation lines which are really necessary to achieve the communication goal of scenario 1 or the coordination goal of scenario 2.

CORSO is seen as an easy to use coordination framework, especially for its transaction capabilities. However, it needs a lot of initial effort (+50%, Table 9) to get the framework to know due to its complex replication strategy and the resulting coordination and notification behavior. For simple scenarios it needs less effort in lines-of-code (LOC) (-5%, Table 9), but when it comes to complex coordination requirements the effort is higher (+20%, Table 9). The complex mapping of application objects to CORSO objects, and the fact that properties of coordination models have to be

represented in the shared object including the payload, implying minor support for real coordination purposes, increases LOC by +25% (Table 9) in comparison to solutions using MozartSpaces.

	Advantages	Limitations
CORSO	• easy to use • transactions • notifications	• mapping of object structure • high initial effort
Java-Spaces	• Fast and reliable • One coordination model • transactions • notifications • low initial effort • Robust against network failures	• Not suitable for complex coordination tasks • High effort needed in case of challenging synchronization and communication requirements • no distributed transactions • no bulk operations
Mozart-Spaces	• transaction support • aspects to extend functionality • good notification support • various coordinators enable e.g., efficient template matching • no garbage collection needed • low time effort for initial training	• difficult management of tuples • border line cases (like empty containers) compound implementation • aspects are hard to implement • careful planning of data structures • bugs • transformation of objects to entries • no distributed transactions
RMI	• good performance • simplicity • minimal lines-of-code • no change between local and remote operations	• low flexibility • high complexity • polling due to missing notification mechanisms • export and unexport of methods • missing transactions • minimal support for synchronizations • adding and change services requires adaptations of processes

Table 8: Summarizing experience reports

JavaSpaces is accepted as a fast and easy to use coordination platform requiring low initial effort demands to learn the Linda like coordination model. However, it is also well accepted that complex coordination requirements cannot be met by the framework only requiring higher efforts (+20%, Table 9) from the software engineer and more LOC (+20%, Table 9) in the application component to satisfy the requirements of the second scenario.

RMI is a popular concept due to its great performance and simplicity. It allows the developer to use remote objects as local ones resulting in minimal LOC (-20% to -40%, Table 9) and effort (-25% to -50%, Table 9) in comparison to MozartSpaces. However, these advantages came along only in case of simple communication requirements. The switch to the second scenario required a lot of effort in time (+40% to +70%, Table 9) and LOC (+30% to +40%, Table 9) from the software developer who had to re-implement issues in the application that are usually supported, like transactions, and not explicitly supported, synchronization between multiple participants, by middleware

concepts. Furthermore, the integration of database to store sessions was an extra effort to be implemented. An interesting finding is the difference between the RMI solutions in sense of time and implementation effort. Depending on the experience of the software engineer the efforts differ by 30% to 40% (Table 9). This may allow the conclusion that non-experienced, average software developers are not supported by the concept explicitly and thus in the course of re-inventing the wheel struggle with implementation details regarding the given requirements.

	CORSO	JavaSpaces	MozartSpaces	RMI
LOC Scen. 1	~ 95%	110%	100%	60% - 80%
LOC Scen. 2	~ 125%	120%	100%	~130% - 160%
Effort Scen. 1	~150%	110%	100%	50% - 75%
Effort Scen. 2	~120%	120%	100%	140% - 170%

Table 9: Reported lines-of-code and effort for comparison of CORSO and RMI with MozartSpaces

In general MozartSpaces is seen as a coordination framework with low initial efforts, good notification and transaction support. The various supported coordination models enable minimal implementations. Aspects are especially well accepted since they allow the extension of operation semantics. However, MozartSpaces has limitations as well implying efforts for work-arounds. Especially when it comes to management of containers and requests regarding the process of coordination itself additional coding has to be done. However, work-arounds can be minimized by a cleaner implementation an enriched functionality of the MozartSpaces software architecture, which can be seen as remaining future work. After refinement of the alpha version of the architecture, a re-evaluation of usability has to be conducted once again.

6.3. *General Discussion*

This section discusses the results of the evaluation of the SBC architectural style with regard to the research issues identified in section 3.1. The findings of the thesis base on the concepts of XVSM (i.e. containers, coordinators, aspects, answer container, and notifications), the combination of architectural styles, and the separation of issues related to distributed systems into categories.

6.3.1. Interactions in Complex Software Systems

C1.1 - Space-Based Computing bridges control- and data-driven coordination: Control-driven coordination models suit best in scenarios with point-to-point or 1:N

communication requirements. Data-driven coordination models on the other hand are effective when several processes need to be synchronized to reach a common goal. As theoretically proven in section 4.3 by means of mapping traditional architectural styles onto the SBC architectural style, it is shown that SBC is capable of representing both coordination models. The style allows software developers to build applications being suitable for both coordination models and to switch between the models requiring small changes (regarding operation parameters) in the implementation of coordinating processes. Due to the variety of changing requirements, as described in section 6.1.2.1 and 6.2, the usage of one coordination model is not sufficient. Under such circumstances application components need to be made aware of both coordination models by typically introducing additional logic bridging them. Based on architectural comparison and empirical evaluation, those sections could show that the SBC architectural style realized by means of XVSM concepts allows application components the transparent usage of both coordination models. Therefore, SBC bridges both control- and data-driven coordination models.

C1.2 - Space-Based Computing improves coordination efficiency: Coordination requirements are reflected in so called coordinators which distinguish between coordination data and payload. The evaluation of benchmark results in sections 6.1.1.1 and 6.1.3.2 show that this distinction improves the efficiency of coordination significantly. This is due to the fact that a coordinator can be implemented efficiently with respect to the context and the coordination requirements of the scenario.

C1.3 - Space-based Computing facilitates continuous coordination: Application components participating in coordination processes may fail because they might have crashed or lost connection to the other participants. In both cases it is necessary that the reintegration of recovered application components is performed fast to facilitate accurate coordination. From the system's point of view application components have recovered if they have a connection to containers representing the coordination process. SBC supports this issue in two ways: replication and aggregation.

SBC stores coordination data in containers which can be transparently replicated, as demonstrated in sections 5.1.4.2 and 5.3.3. Transparent replication however facilitates a global view of all participating application components of the coordination process. Section 6.1.3.1 compares the capabilities of the XVSM based architecture with traditional architectures by measuring execution time and number of processing steps. The results say, that in case an application component crashes, the reintegration of that component is performed transparent, is integrated into the coordination process with zero-delay, and needs fewer processing steps when using XVSM.

In section 5.1.4.3 it is shown that XVSM concepts allow the outsourcing of specific business logic enabling the aggregation of messages while the application component is offline. The strategy is to minimize the number of messages to be retrieved when the application component recovers by reestablishing a connection to the system. Section

6.1.1.3 performs architecture comparison, counts the number of aggregated messages, and the time for delivery. The outcome is that the aggregation of messages facilitates to receive only those messages which are relevant for efficient coordination.

C1.4 - Space-based Computing reduces coordination complexity in applications: The concept of coordinators in containers moves the complexity of coordination requirement away from application components to a central point in the SBC coordination framework. The complexity of a coordination issue is concentrated at one point enabling a clear separation between business logic and coordination logic again. Sections 6.1.1.4, 6.1.3.1 and 6.1.3.2 compare the number of processing steps needed to realize a coordination requirement or resp. discuss theoretically the impact of different architectures. The results show that by moving the complexity into the coordinator coordination requirements can be reduced to a single operation call on a container. Additionally, since coordination inherently consists of communication, aspects of communication can be abstracted as well by reducing the number of operations to a minimum.

6.3.2. Evolution of Complex Software Systems

C2.1 - Space-based Computing supports flexible software architectures: SBC framework explicitly distinguishes between five XVSM categories each responsible for managing specific complexity issues of distributed environments. This helps system developers to identify only those components which are affected by a new business requirement, and thus minimize the effects of the implementation of the new requirement on other categories. The explicit categorization guides average software developers in making design and implementation decisions. This means that in case of design mistakes and implementation errors the effects of the decision is limited to the category where it has been made. By comparing traditional architectures and XVSM based architectures in sections 6.1.2.2 and 6.1.1.2, it is shown that SBC enables the change of functional and non-functional properties in software architecture transparent to application components. Empirical evaluation in section 6.2 shows that XVSM's flexibility is well recognized by software developers being prepared for future requirements.

C2.2 - Space-based Computing facilitates robustness against changing requirements: The SBC architectural style is capable of representing the combined power of several architectural styles (i.e. dataflow, repository, explicit, and implicit architectural styles) with the same number of SBC concepts using a standardized basic API. Architectural comparisons in sections 6.1.2.1, 6.1.1.2, and 6.1.3.1 show that XVSM based architectures are more robust against changing requirements in comparison to traditional architectures. Communication and Extension Profiles also facilitate the implementation of new requirements without changing the general

architecture of the system. Empirical evaluation in section 6.2 shows that in case of simple communication requirements, traditional message-based approaches are sufficient. However, if sophisticated coordination capabilities have to be realized the complexity of the application and the time needed to implement those requirements increases effort and time in comparison to architectural solutions using SBC.

C2.3 - Space-based Computing reduces complexity in applications introduced due to changing requirements:

The five XVSM categories of decoupling demonstrate where and how various aspects related to coordination and distribution can be realized, introducing a “separation of concerns” between the categories. Sections 6.1.1.2, 6.1.2, 6.1.3.1, and 6.2 evaluated the effects on the complexity of application components by discussing the result of the introduction of common changes. The outcome is that separation of concerns allows software developers to introduce new requirements in various categories without affecting application components.

Chapter 7

7. Conclusion and Perspectives

Today's software systems can be seen as complex systems in sense that they usually interact with other software, systems, devices, sensors and people over distributed, heterogeneous, decentralized and interdependent environments while operated more often in dynamic and frequently unpredictable circumstances. Therefore, software developers have to deal with issues like heterogeneity and varying size of components, variety of protocols for interaction with internal and external components, number of potential incidents, like crashed or unreachable components in distributed environments, or adaptability of the system throughout its lifetime. Those software systems typically consist of mainly distributed application components representing higher-level business goals and a middleware technology usually representing an architectural style and abstracting the complexity concerns related to network and distribution.

With respect to complex software systems, this work focuses on the question how interaction capabilities between application components can be improved and how dependency on architectural styles can be diminished while minimizing the complexity of the software architecture in case of changes.

The message-passing paradigm is a common concept allowing application components to interact with each other. But even asynchronous message-oriented middleware technologies are not suitable for complex coordination requirements since the processing and state of coordination have to be handled explicitly by the application component, thus increasing its complexity. Data-driven frameworks, like tuple spaces, support the coordination of application components, but have a limited number of coordination policies. Therefore, with respect to more complex coordination requirements application components still need to implement coordination functionality that is not directly supported by the coordination framework.

The dependency between application components and architectural styles comes forward when changes regarding the software architecture have to be realized. In case a new business requirement demands the implementation of other architectural styles, the combinations of those styles further introduces additional cognitive complexity. Consequently, instead of a stable set of architectural concepts for effectively managing complexity concerns, the number of concepts a software developer has to work with explicitly increases with the size and degree of evolution of the system.

The thesis proposes the Space-Based Computing architectural style that flexibly combines and abstracts the properties of several architectural styles and extends them by sophisticated coordination models. The Space-Based Computing architectural style

has been realized in the eXtensible Virtual Shared Memory (XVSM) reference architecture consisting of the concepts container, coordinator, and aspect accessed via a simple API. The concepts are categorized into five categories (computation, coordination, organization, distribution, and communication) enabling to cope with different aspects of complexity of distributed systems in one of the categories with minimal mutual influence.

The research results were evaluated in three industrial application domains in air traffic management, production automation, and intelligent transportation regarding feasibility, effort, robustness, performance, and usability. The evaluation is based on prototypes for a set of specific use cases of the industrial application domains, as well as on empirical studies. The evaluation was carried out by means of prototype implementations, studies, benchmarks, and theoretical proofs. Major results of this work are a higher coordination efficiency, improved robustness against changing requirements, simplified realization of business requirements, and reduced complexity in applications.

Remaining future work refers to research topics such as the improvement of evaluation strategies for complexity measurement, investigation of scenarios with high-frequently changing conditions both of infrastructure and application requirements and capabilities, and wide-scale benchmarks of the proposed reference architecture with respect to scalability.

References

1. Aberer, K. *P-Grid: A Self-Organizing Access Structure for P2P Information Systems*. in *CooplS '01: Proceedings of the 9th International Conference on Cooperative Information Systems*. 2001. London, UK: Springer-Verlag.
2. ActiveSpace. *WebSite*. 2010 [cited; Available from: http://activespace.codehaus.org/.
3. Aladejana, F., *The Implications of ICT and NKS for Science Teaching: Whither Nigeria.* Complex Systems-Champaign-, 2007. **17**(1/2): p. 113.
4. Albert, J. and D. Abhijit, *Test beds for complex systems.* Commun. ACM, 2005. **48**(5): p. 45-50.
5. Allen, M., *Novell IPX over Various WAN Media (IPXWAN).* 1992: RFC Editor.
6. Alvisi, L. and K. Marzullo, *Message Logging: Pessimistic, Optimistic, Causal, and Optimal.* IEEE Trans. Softw. Eng., 1998. **24**(2): p. 149--159.
7. Amaral, L. and J. Ottino, *Complex networks: Augmenting the framework for the study of complex systems.* The European Physical Journal B-Condensed Matter, 2004. **38**(2): p. 147-162.
8. Anand, R. and H.C. Roy, *What is the complexity of a distributed computing system?* Complexity, 2007. **12**(6): p. 37-45.
9. Arbab, F., I. Herman, and P. Spilling, *Manifold: Concepts and Implementation*, in *Proceedings of the Second Joint International Conference on Vector and Parallel Processing: Parallel Processing*. 1992, Springer-Verlag.
10. Auprasert, B. and Y. Limpiyakorn, *Structuring Cognitive Information for Software Complexity Measurement*, in *Proceedings of the 2009 WRI World Congress on Computer Science and Information Engineering - Volume 07*. 2009, IEEE Computer Society.
11. Avgeriou, P. and U. Zdun. *Architectural Patterns Revisited - A Pattern Language*. in *Proc. Of 10th European Conference on Pattern Languages of Programs (EuroPLoP 2005)*. 2005.
12. Bai, X., J. Xie, B. Chen, and S. Xiao. *DRESR: Dynamic Routing in Enterprise Service Bus*. in *ICEBE '07: Proc. of the IEEE Int. Conf. on e-Business Engineering*. 2007. Washington, DC, USA: IEEE Computer Society.
13. Balasubramanian, K., A. Gokhale, G. Karsai, J. Sztipanovits, and S. Neema, *Developing Applications Using Model-Driven Design Environments.* Computer, 2006. **39**(2): p. 33.
14. Balazinska, M., H. Balakrishnan, and D. Karger, *INS/Twine: A Scalable Peer-to-Peer Architecture for Intentional Resource Discovery.* 2002: p. 195--210.
15. Bansal, A.K., K. Ramamohanarao, and A.S. Rao. *Distributed Storage of Replicated Beliefs to Facilitate Recovery of Distributed Intelligent Agents*. in *ATAL '97: Proceedings of the 4th International Workshop on Intelligent Agents IV, Agent Theories, Architectures, and Languages*. 1998. London, UK: Springer-Verlag.

16. Barbara, A.K., P. Shari Lawrence, M.P. Lesley, W.J. Peter, C.H. David, E. Khaled El, and R. Jarrett, *Preliminary guidelines for empirical research in software engineering.* IEEE Trans. Softw. Eng., 2002. **28**(8): p. 721-734.
17. Barrientos Garcia, F.J., I.G. incertis, F.M. Trespaderne, E.F. Lopez, and J.R. Peran Gonzalez. *System for the production control and automatic packaging of plastic air sleeve guides*. in *Engineering Complex Computer Systems, 2004. Proceedings. Ninth IEEE International Conference on*. 2004.
18. Beck, K. and C. Andres, *Extreme Programming Explained: Embrace Change (2nd Edition)*. 2004: Addison-Wesley Professional.
19. Bessler, S., A. Fischer, E. Kühn, R. Mordinyi, and S. Tomic, *Using Tuple-Spaces to manage the Storage and Dissemination of Spatial-temporal Content.* Journal of Computer and System Sciences, 2010.
20. Bettini, L., V. Bono, R.D. Nicola, G. Ferrari, D. Gorla, M. Loreti, E. Moggi, R. Pugliese, E. Tuosto, and B. Venneri, *The Klaim Project: Theory and Practice*. 2003.
21. Bhola, S., R. Strom, S. Bagchi, Y. Zhao, and J. Auerbach. *Exactly-once Delivery in a Content-based Publish-Subscribe System*. in *DSN*. 2002.
22. Biffl, S., E. Kühn, A. Schatten, R. Mordinyi, and T. Moser, *Linking Contributions of Informatics Research to Complex Systems Challenges.* Technical Report, Vienna University of Technology, 2009.
23. Biffl, S., R. Mordinyi, and A. Schatten. *A Model-Driven Architecture Approach Using Explicit Stakeholder Quality Requirement Models for Building Dependable Information Systems*. in *Fifth International Workshop on Software Quality (WoSQ'07). ICSE Workshops 2007*. 2007.
24. Blieberger, J., J. Klasek, e. Kühn, and hn, *Ada Binding to a Shared Object Layer*, in *Proceedings of the 1999 Ada-Europe International Conference on Reliable Software Technologies*. 1999, Springer-Verlag.
25. Bob, C., *Complexity in Design.* IEEE Computer, 2005. **38**(10): p. 10-12.
26. Braccesi, L., M. Monsignori, and P. Nesi. *Monitoring and optimizing industrial production processes*. in *Engineering Complex Computer Systems, 2004. Proceedings. Ninth IEEE International Conference on*. 2004.
27. Brereton, P., B.A. Kitchenham, D. Budgen, M. Turner, and M. Khalil, *Lessons from applying the systematic literature review process within the software engineering domain.* J. Syst. Softw., 2007. **80**(4): p. 571-583.
28. Brooks, R., *I, Rodney Brooks, Am a Robot.* IEEE Spectrum Online, 2008.
29. Broy, M., *The 'Grand Challenge' in Informatics: Engineering Software-Intensive Systems.* Computer, 2006. **39**(10): p. 72-80.
30. Buschmann, F., K. Henney, and D. Schmidt, *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing (Wiley Software Patterns Series)*. 2007: John Wiley \& Sons.
31. Butler, R.M., A.L. Leveton, and E.L. Lusk, *p4-Linda: a portable implementation of Linda.* High Performance Distributed Computing, 1993., Proceedings the 2nd International Symposium on, 1993: p. 50-58.
32. Cabri, G., L. Leonardi, G. Reggiani, and F. Zambonelli, *Design and implementation of a programmable coordination architecture for mobile agents.* Technology of Object-Oriented Languages and Systems, 1999. Proceedings of, 1999: p. 10-19.

33. Cabri, G., L. Leonardi, and F. Zambonelli. *Reactive Tuple Spaces for Mobile Agent Coordination*. in *MA '98: Proceedings of the Second International Workshop on Mobile Agents*. 1998. London, UK: Springer-Verlag.
34. Cabri, G., L. Leonardi, and F. Zambonelli, *MARS: a programmable coordination architecture for mobile agents.* Internet Computing, IEEE, 2000. **4**(4): p. 26-35.
35. Cabri, G., L. Leonardi, and F. Zambonelli, *Mobile Agent Coordination for Distributed Network Management.* Journal of Network and Systems Management, 2001. **9**(4): p. 435--456.
36. Capizzi, S., *A Tuple Space Implementation for Large-Scale Infrastructures*. 2008, Phd Thesis, Department of Computer Science, University of Bologna.
37. Carzaniga, A., *Architectures for an Event Notification Service Scalable to Wide-area Networks*. 1998, Politecnico Di Milano.
38. Carzaniga, A., E.D. Nitto, D.S. Rosenblum, and A.L. Wolf. *Issues in supporting event-based architectural styles*. in *ISAW '98: Proceedings of the third international workshop on Software architecture*. 1998. New York, NY, USA: ACM Press.
39. Carzaniga, A., D.S. Rosenblum, and A.L. Wolf, *Design and evaluation of a wide-area event notification service.* ACM Transactions on Computer Systems, 2001. **19**: p. 332--383.
40. Chappell, D., *Enterprise Service Bus*. 2004: O'Reilly Media, Inc.
41. Christian, G., H. Hans-Michael, and E. Sven, *From IEC 61131 to IEC 61499 for distributed systems: a case study.* EURASIP J. Embedded Syst., 2008. **2008**: p. 1-8.
42. Ciancarini, P. *PoliS: a programming model for multiple tuple spaces*. in *IWSSD '91: Proceedings of the 6th international workshop on Software specification and design*. 1991. Los Alamitos, CA, USA: IEEE Computer Society Press.
43. Ciancarini, P., *Coordination models and languages as software integrators.* ACM Comput. Surv., 1996. **28**(2): p. 300--302.
44. Ciancarini, P., K. Jensen, and D. Yankelevich, *On the operational semantics of a coordination language*, in *Object-Based Models and Languages for Concurrent Systems*. 1995. p. 77-106.
45. Ciancarini, P., M. Mazza, and L. Pazzaglia, *A logic for a coordination model with multiple spaces.* Sci. Comput. Program., 1998. **31**(2-3): p. 231--261.
46. Cilliers, P., *Complexity and Postmodernism: Understanding Complex Systems*. 1998: Routledge.
47. Cohen, D., M. Lindvall, and P. Costa, *An introduction to agile methods.* Advances in Computers, 2004. **62**: p. 2--67.
48. Committee, O.W.T. *Web services business process execution language version 2.0.* . OASIS Committee Specification 2007 [cited; Available from: http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html.
49. Costa, C.A.d., A.C. Yamin, and C.F.R. Geyer, *Toward a General Software Infrastructure for Ubiquitous Computing*. 2008. p. 64-73.
50. Crass, S., *A Formal Model of the Extensible Virtual Shared Memory (XVSM) and its Implementation in Haskell*. 2010, Institute of Computer Languages, Vienna University of Technology.
51. Craß, S., E. Kühn, and G. Salzert. *Algebraic foundation of a data model for an extensible space-based collaboration protocol*. in *Proceedings of the 2009*

International Database Engineering \&\#38; Applications Symposium (IDEAS 2009). 2009. New York, NY, USA: ACM.

52. Cremonini, M., A. Omicini, and F. Zambonelli, *Coordination and Access Control in Open Distributed Agent Systems: The TuCSoN Approach*. 2000.
53. Cruz, J.C. and S. Ducasse, *A Group Based Approach for Coordinating Active Objects*, in *Proceedings of the Third International Conference on Coordination Languages and Models*. 1999, Springer-Verlag.
54. Cugola, G., E. Di Nitto, and A. Fuggetta, *The JEDI Event-Based Infrastructure and Its Application to the Development of the OPSS WFMS.* IEEE Trans. Softw. Eng., 2001. **27**(9): p. 827--850.
55. Dabek, F., P. Druschel, B. Zhao, J. Kubiatowicz, and I. Stoica, *Towards a Common API for Structured Peer-to-Peer Overlays*, in *Proceedings of the 2nd International Workshop on Peer-to-peer Systems (IPTPS'03)*. 2003.
56. David, G., A. Robert, and O. John, *Architectural mismatch or why it's hard to build systems out of existing parts*, in *Proceedings of the 17th international conference on Software engineering*. 1995, ACM: Seattle, Washington, United States.
57. Denti, E., A. Natali, and A. Omicini, *Programmable Coordination Media*, in *Proceedings of the Second International Conference on Coordination Languages and Models*. 1997, Springer-Verlag.
58. Denti, E. and A. Omicini, *An architecture for tuple-based coordination of multi-agent systems.* Softw. Pract. Exper., 1999. **29**(12): p. 1103-1121.
59. Denti, E., A. Omicini, and V. Toschi, *Coordination Technology for the Development of Multi-Agent Systems on the Web.* Proceedings of the 6th AI*IA Congress of the Italian Association for Artificial Intelligence (AI*IA'99), 1999: p. 29-38.
60. Deutsch, P. *The Eight Fallacies of Distributed Computing*. 2005 [cited; Available from: http://blogs.sun.com/jag/resource/Fallacies.html.
61. Dingsoyr, T. and T. Dyba, *What Do We Know about Agile Software Development?* Software, IEEE, 2009. **26**(5): p. 6-9.
62. Dustdar, S., H. Gall, and M. Hauswirth, *Software-Architekturen für Verteilte Systeme*. 2003: Xpert.press.
63. Elisabetta Di, N. and R. David, *Exploiting ADLs to specify architectural styles induced by middleware infrastructures*, in *Proceedings of the 21st international conference on Software engineering*. 1999, ACM: Los Angeles, California, United States.
64. Elnozahy, E.N., L. Alvisi, Y.-M. Wang, and D.B. Johnson, *A survey of rollback-recovery protocols in message-passing systems.* ACM Comput. Surv., 2002. **34**(3): p. 375--408.
65. Emmott, S. and S. Rison, *Towards 2020 Science*. 2006, Microsoft Research.
66. Erickson, J., K. Lyytinen, and K. Siau, *Agile Modeling, Agile Software Development, and Extreme Programming: The State of Research.* Journal of Database Management, 2005. **16 (4)**: p. 88 - 99.
67. Eugster, P., *Type-based publish/subscribe: Concepts and experiences.* ACM Trans. Program. Lang. Syst., 2007. **29**(1): p. 6.
68. Eugster, P.T., P.A. Felber, R. Guerraoui, and A.M. Kermarrec, *The many faces of publish/subscribe.* ACM Comput. Surv., 2003. **35**(2): p. 114--131.

69. eva, K., hn, and N. Georg, *Post-Client/Server Coordination Tools*, in *Coordination Technology for Collaborative Applications - Organizations, Processes, and Agents [ASIAN 1996 Workshop]*. 1998, Springer-Verlag.
70. Fiege, L., *Visibility in Event-Based Systems*. 2004, Technischen UniversitÃ¤t Darmstadt.
71. Floyd, C., *A systematic look at prototyping.* Approaches to prototyping, 1984: p. 1-18.
72. Francois, A.R., *Software architecture for computer vision: Beyond pipes and filters*. 2003, Technical Report IRIS-03-240, Institute for Robotics and Intelligent Systems, USC.
73. Franklin, S., *Coordination without Communication*. 2008, Inst. For Intelligent Systems, Univ. of Memphis.
74. Freeman, E., K. Arnold, and S. Hupfer, *JavaSpaces Principles, Patterns, and Practice*. 1999, Essex, UK, UK: Addison-Wesley Longman Ltd.
75. Gagliardi, F. and F. Grey, *Old World, New Grid* IEEE Spectrum Online, 2006.
76. Gail, W.B., *Climate Control.* IEEE Spectrum Online, 2007.
77. Garlan, D. and M. Shaw, *An Introduction to Software Architecture*. 1994, Carnegie Mellon University.
78. Gelernter, D., *Generative communication in Linda.* ACM Trans. Program. Lang. Syst., 1985. **7**(1): p. 80--112.
79. Gelernter, D. and N. Carriero, *Coordination languages and their significance.* Commun. ACM, 1992. **35**(2): p. 96.
80. Gershenson, C., *Self-organizing traffic lights.* Complex Systems, 2005.
81. Gibaud, A. and P. Thomin, *Communications directed by bound types in Linda: presentation and formal model.* Parallel and Distributed Systems, IEEE Transactions on, 2002. **13**(8): p. 828-843.
82. Gilanyi, Z., *Modelling Markets Versus Market Economies: Success and Failure.* Interdisciplinary Description of Complex Systems, 2005. **3**(2): p. 94--99.
83. Gilanyi, Z., *Some Remarks on the Effects of Productivity on Growth.* Interdisciplinary Description of Complex Systems, 2007. **5**(1): p. 14--20.
84. Gilb, T., *Competitive Engineering: A Handbook For Systems Engineering, Requirements Engineering, and Software Engineering Using Planguage*. 2005, Newton, MA, USA: Butterworth-Heinemann.
85. Goiss, H.-D., *Naming and Communication Management for MozartSpaces - Using Space Based Computing on a Wider Scale*. 2009, Institute of Computer Languages, Vienna University of Technology.
86. Graff, D., R. Menezes, and R. Tolksdorf, *On the performance of swarm-based tuple organization in LINDA systems.* Evolutionary Computation, 2008. CEC 2008. (IEEE World Congress on Computational Intelligence). IEEE Congress on, 2008: p. 2709-2716.
87. Grandpierre, A., *Complexity, Information and Biological Organisation.* Interdisciplinary Description of Complex Systems, 2005. **3**(2): p. 59--71.
88. Gündüz, G., *Ancient and Current Chaos Theories.* Interdisciplinary Description of Complex Systems, 2006. **4**(1): p. 1--18.
89. Hadar, E. and G.M. Silberman. *Agile architecture methodology: long term strategy interleaved with short term tactics*. in *OOPSLA Companion '08: Companion to the 23rd ACM SIGPLAN conference on Object-oriented*

programming systems languages and applications. 2008. New York, NY, USA: ACM.
90. Halevy, A., *Why Your Data Won't Mix.* Queue, 2005. **3**(8): p. 50--58.
91. Harri, H. and T. Antti, *HSDPA/HSUPA for UMTS: High Speed Radio Access for Mobile Communications*. 2006: John Wiley \& Sons.
92. Harris, G.E., D. Leibs, r.J. Carri\`e, F. Nagy, J. Crupi, and M. Nally. *Application servers: one size fits all...not?* in *OOPSLA'03: Companion of the 18th annual ACM SIGPLAN Conf. on Object-oriented programming, systems, languages, and applications*. 2003: ACM.
93. Hazra, T.K., *Occam channels and Kernel Linda.* Potentials, IEEE, 1993. **12**(1): p. 15-17.
94. Henning, M., *The Rise and Fall of CORBA.* Queue, 2006. **4**(5): p. 28-34.
95. Highsmith, J. and A. Cockburn, *Agile software development: the business of innovation.* Computer, 2001. **34**(9): p. 120-127.
96. Hirmer, S., H. Kaiser, A. Merzky, A. Hutanu, and G. Allen, *Generic support for bulk operations in grid applications*, in *Proceedings of the 4th international workshop on Middleware for grid computing*. 2006, ACM: Melbourne, Australia.
97. Hirsch, M., C. Gerber, H.M. Hanisch, and V. Vyatkin. *Design and Implementation of Heterogeneous Distributed Controllers According to the IEC 61499 Standard - A Case Study*. in *Industrial Informatics, 2007 5th IEEE International Conference on*. 2007.
98. Hoare, T. and R. Miller, *Grand Challenges in Computing - Research*. 2004, The British Computer Society.
99. Hobfeld, T., S. Oechsner, K. Tutschku, F.-U. Andersen, and L. Caviglione, *Supporting Vertical Handover by Using a Pastry Peer-to-Peer Overlay Network*, in *Proceedings of the 4th annual IEEE international conference on Pervasive Computing and Communications Workshops*. 2006, IEEE Computer Society.
100. Hoekstra, R.C., H. van Arkel, and B. Leurs, *Modeling Local Monetary Flows in Poor Regios: A Research Setup To Simulate the Multiplier Effect in Local Economies.* Interdisciplinary Description of Complex Systems, 2007. **5**(2): p. 138--150.
101. Hohpe, G. and B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. 2003, Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.
102. Howard, F.L., R.M. Nancy, and P.M. Andrew, *Can We Ever Build Survivable Systems from COTS Components?*, in *Proceedings of the 14th International Conference on Advanced Information Systems Engineering*. 2002, Springer-Verlag.
103. Huang, Y. and H. Garcia-Molina, *Publish/subscribe in a mobile environment.* Wirel. Netw., 2004. **10**(6): p. 643--652.
104. Jang, M.-W., A.A. Momen, and G. Agha. *ATSpace: A Middle Agent to Support Application Oriented Matchmaking and Brokering Services*. in *IAT '04: Proceedings of the Intelligent Agent Technology, IEEE/WIC/ACM International Conference*. 2004. Washington, DC, USA: IEEE Computer Society.
105. Jayadev, M., *Computation Orchestration - A basis for wide-area computing*, in *Engineering Theories of Software Intensive Systems*. 2005. p. 285-330.

106. Jazayeri, M. *On Architectural Stability and Evolution*. in *da-Europe '02: Proceedings of the 7th Ada-Europe International Conference on Reliable Software Technologies*. 2002. London, UK: Springer-Verlag.
107. Jeong, K. and D. Shasha, *PLinda 2.0: a transactional/checkpointing approach to fault tolerant Linda.* Reliable Distributed Systems, 1994. Proceedings., 13th Symposium on, 1994: p. 96-105.
108. Jiang, Y., G. Xue, Z. Jia, and J. You, *DTuples: A Distributed Hash Table based Tuple Space Service for Distributed Coordination.* Grid and Cooperative Computing, 2006. GCC 2006. Fifth International Conference, 2006: p. 101-106.
109. Jorgensen, H., O. Lawrence, and A. Neus. *Incessant change is the norm.* IBM Global Business Services 2010 [cited; Available from: http://www-935.ibm.com/services/us/gbs/bus/html/gbs-making-change-work.html.
110. Jose-Norberto, M., T. Juan, S. Manuel, and P. Mario, *Applying MDA to the development of data warehouses*, in *Proceedings of the 8th ACM international workshop on Data warehousing and OLAP*. 2005, ACM: Bremen, Germany.
111. Joseph, T., H.J. Wen, and A. Neveen, *Health care and services delivery systems as complex adaptive systems.* Commun. ACM, 2005. **48**(5): p. 36-44.
112. Karhinen, A., J. Kuusela, and T. Tallgren. *An architectural style decoupling coordination, computation and data*. in *Engineering of Complex Computer Systems, 1997. Proceedings., Third IEEE International Conference on*. 1997.
113. Karolus, M., *Design and Implementation of XcoSpaces, the .Net Reference Implementation of XVSM - Coordination, Transactions and Communication* 2009, Institute of Computer Languages, Vienna University of Technology.
114. Kemppainen, K., *Priority scheduling revisited - dominant rules, open protocols and integrated order management*. 2005, Acta Universitatis oeconomicae Helsingiensis. A.
115. Keszthelyi, L., *Design and Implementation of the JavaSpaces API Standard for XVSM*. 2008, Institute of Computer Languages, Vienna University of Technology.
116. Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, *Aspect-oriented programming*. 1997.
117. Kinga, M. and C. Adrian, *GLinda - Grid-Based Distributed Linda System.* Symbolic and Numeric Algorithms for Scientific Computing, 2007. SYNASC. International Symposium on, 2007: p. 349-352.
118. Kolmann, P., *University campus grid computing*. 2005, Vienna University of Technology.
119. Kubalik, J. and R. Mordinyi. *Optimizing Events Traffic in Event-based Systems by means of Evolutionary Algorithms*. in *The Second International Conference on Availability, Reliability and Security (ARES 2007)*. 2007.
120. Kubalik, J., R. Mordinyi, and S. Biffl, *Multiobjective Prototype Optimization with Evolved Improvement Steps.* Evolutionary Computation in Combinatorial Optimization, 2008: p. 218--229.
121. Kühn, E., *Implementierung von Multi-Datenbanksystemen in Prolog*. 1989, Phd Thesis, Vienna University of Technology.
122. Kühn, E., *Virtual Shared Memory for Distributed Architecture*. 2001: Nova Science Publishers.
123. Kühn, E., R. Mordinyi, H.D. Goiss, S. Bessler, and S. Tomic, *Integration of Shareable Containers with Distributed Hash Tables for Storage of Structured*

and Dynamic Data. Proceedings of the 2nd International Workshop on Adaptive Systems in Heterogeneous Environments (ASHEs 2009), 2009.

124. Kühn, E., R. Mordinyi, L. Keszthelyi, and C. Schreiber. *Introducing the concept of customizable structured spaces for agent coordination in the production automation domain*. in *Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009)*. 2009. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems.
125. Kühn, E., R. Mordinyi, L. Keszthelyi, C. Schreiber, S. Bessler, and S. Tomic, *Aspect-oriented Space Containers for Efficient Publish/Subscribe Scenarios in Intelligent Transportation Systems.* Proceedings of the 11th International Symposium on Distributed Objects, Middleware, and Applications (DOA'09), 2009.
126. Kühn, E., R. Mordinyi, M. Lang, and A. Selimovic, *Towards Zero-Delay Recovery of Agents in Production Automation Systems.* IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (IAT 2009), 2009. **2**: p. 307-310.
127. Kühn, E., R. Mordinyi, and C. Schreiber, *Configurable Notifications for Event-based Systems*. 2008, Vienna University of Technology, (TechRep at http://tinyurl.com/oht888).
128. Kühn, E., R. Mordinyi, and C. Schreiber, *An Extensible Space-based Coordination Approach for Modeling Complex Patterns in Large Systems.* 3rd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, Special Track on Formal Methods for Analysing and Verifying Very Large Systems, 2008.
129. Larry, P., *Personal computing: simple complexity and COMDEX.* Commun. ACM, 1990. **33**(7): p. 21-26.
130. Lehman, T.J., A. Cozzi, Y. Xiong, J. Gottschalk, V. Vasudevan, S. Landis, P. Davis, B. Khavar, and P. Bowman, *Hitting the distributed computing sweet spot with TSpaces.* Comput. Netw., 2001. **35**(4): p. 457--472.
131. Lehman, T.J., S.W. McLaughry, and P. Wycko. *T-Spaces: The Next Wave*. in *Hawaii Intl. Conf. on System Sciences (HICSS-32)*. 1999.
132. Leu, P.-J. and B.K. Bhargava. *Concurrent Robust Checkpointing and Recovery in Distributed Systems*. in *Proceedings of the Fourth International Conference on Data Engineering*. 1988. Washington, DC, USA: IEEE Computer Society.
133. Li, H.F., Z. Wei, and D. Goswami, *Quasi-atomic recovery for distributed agents.* Parallel Comput., 2006. **32**(10): p. 733--758.
134. Liu, F.C.S., *Constrained Opinion Leader Influence in an Electoral Campaign Season: Revisiting the Two-Step Flow Theory with Multi-Agent Simulation.* Advances in Complex Systems, 2007. **10**(2): p. 233.
135. Liu, Y. and B. Plale, *Survey of Publish Subscribe Event Systems*. 2003, Computer Science Deptartment, Indiana University.
136. Luckham, D.C., *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. 2001, Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.
137. Lüder, A., J. Peschke, T. Sauter, S. Deter, and D. Diep, *Distributed intelligence for plant automation based on multi-agent systems: the PABADIS approach.* Production Planning and Control, 2004. **15**: p. 201-212.

138. Malone, T.W., *What is coordination theory?* 1988, Cambridge, MA: MIT Sloan School of Management.
139. Malone, T.W. and K. Crowston. *What is coordination theory and how can it help design cooperative work systems?* in *CSCW '90: Proceedings of the 1990 ACM conference on Computer-supported cooperative work*. 1990. New York, NY, USA: ACM.
140. Malone, T.W. and K. Crowston, *The interdisciplinary study of coordination.* ACM Comput. Surv., 1994. **26**(1): p. 87--119.
141. Marek, A., *Design and Implementation of TinySpaces - The .NET Micro Framework based Implementation of XVSM for Embedded Systems*. 2010, Institute of Computer Languages, Vienna University of Technology.
142. Mark, K., S. Hiroki, F. Peyman, and B.-Y. Yaneer, *A complex systems perspective on computer-supported collaborative design technology.* Commun. ACM, 2002. **45**(11): p. 27-31.
143. McBreen, P., *Questioning Extreme Programming*. 2002, Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.
144. McDermid, J.A. *Complexity: Concept, Causes and Control*. in *6th IEEE international Conference on Complex Computer Systems*. 2000: IEEE Computer Society.
145. Menezes, R. and R. Tolksdorf. *A new approach to scalable Linda-systems based on swarms*. in *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*. 2003. New York, NY, USA: ACM.
146. Merdan, M., T. Moser, D. Wahyudin, and S. Biffl. *Performance evaluation of workflow scheduling strategies considering transportation times and conveyor failures*. 2008.
147. Merdan, M., T. Moser, D. Wahyudin, S. Biffl, and P. Vrba, *Simulation of Workflow Scheduling Strategies Using the MAST Test Management System.* 10th International Conference on Control, Automation, Robotics and Vision, 2008.
148. Miller, J. and J. Mukerji. *Model Driven Architecture (MDA).* January 2007 [cited; Available from: http://www.omg.org/docs/ormsc/01-07-01.pdf.
149. Mohtashemi, M., B.W. Higgs, and R. Levins, *Infection and Atherosclerosis: Is There an Association?* Complex Systems-Champaign-, 2006. **16**(3): p. 259.
150. Monson-Haefel, R. and D. Chappell, *Java Message Service*. 2000: O'Reilly & Associates, Inc. 220.
151. Mordinyi, R., E. Kühn, and A. Schatten. *An Architectural Framework for Agile Software Development*. in *Proceedings of the 11th International Conference on Agile Software Development (XP 2010).*
152. Mordinyi, R., T. Moser, E. Kühn, S. Biffl, and A. Mikula, *Foundations for a Model-Driven Integration of Business Services in a Safety-Critical Application Domain.* Software Engineering and Advanced Applications, Euromicro Conference, 2009. **0**: p. 267-274.
153. Moser, T., *Semantic Integration of Engineering Environments Using an Engineering Knowledge Base*. 2010, Phd Thesis, Vienna University of Technology.
154. Moser, T., R. Mordinyi, S. Biffl, and A. Mikula. *Efficient System Integration using Semantic Requirements and Capability Models - An Approach for Integrating Heterogeneous Business Services*. in *Proceedings of the 11th*

International Conference on Enterprise Information Systems (ICEIS), Volume DISI, Milan, Italy, May 6-10, 2009. 2009.

155. Moser, T., R. Mordinyi, A. Mikula, and S. Biffl, *Making Expert Knowledge Explicit to Facilitate Tool Support for Integrating Complex Information Systems in the ATM Domain.* Proceedings of the 2009 International Confernce on Complex, Intelligent and Software Intensive Systems (CISIS 2009), 2009. **0**: p. 90-97.
156. Moser, T., R. Mordinyi, W.D. Sunindyo, and S. Biffl. *Semantic Service Matchmaking in the ATM Domain Considering Infrastructure Capability Constraints*. in *Proceedings of the 21st International Conference on Software Engineering & Knowledge Engineering (SEKE'2009), Boston, Massachusetts, USA, July 1-3, 2009*. 2009.
157. Moser, T., K. Schimper, R. Mordinyi, and A. Anjomshoaa, *SAMOA - A Semi-Automated Ontology Alignment Method for Systems Integration in Safety-Critical Environments.* Proceedings of the 2009 International Confernce on Complex, Intelligent and Software Intensive Systems (CISIS 2009), 2009. **0**: p. 724-729.
158. Murphy, A.L., G.P. Picco, and G.C. Roman, *LIME: A coordination model and middleware supporting mobility of hosts and agents.* ACM Trans. Softw. Eng. Methodol., 2006. **15**(3): p. 279--328.
159. Nancy, G.L., *Safety as a system property.* Commun. ACM, 1995. **38**(11): p. 146.
160. Neelakanta, P.S., M. Leesirikul, Z. Roth, and S. Morgera, *A Complex System Model of Glucose Regulatory Metabolism.* Complex Systems-Champaign-, 2006. **16**(4): p. 343.
161. Nerur, S., R. Mahapatra, and G. Mangalaraj, *Challenges of migrating to agile methodologies.* Commun. ACM, 2005. **48**(5): p. 72--78.
162. Neumann, M., *Complexity of Social Stability: A Model-to-model Analysis of Yugoslavia's Decline.* Interdisciplinary Description of Complex Systems, 2007. **5**(2): p. 92--111.
163. Nicholas, C., G. David, and D.Z. Lenore, *Bauhaus Linda*, in *Selected papers from the ECOOP'94 Workshop on Models and Languages for Coordination of Parallelism and Distribution, Object-Based Models and Languages for Concurrent Systems*. 1995, Springer-Verlag.
164. Nicola, R.D., *Coordination and Access Control of Mobile Agents*. 1999.
165. Nicola, R.D., G.L. Ferrari, and R. Pugliese, *KLAIM: a kernel language for agents interaction and mobility.* Software Engineering, IEEE Transactions on, 1998. **24**(5): p. 315-330.
166. Nicola, R.D. and M. Loreti, *A Modal Logic for Klaim*. 2000.
167. Nielsen, J., *Adapting the Siena Content-Based Publish-Subscribe system to support user mobility*. 2004, Rutgers University - ECE department.
168. Obermaisser, R. and H.Kopetz, *GENESYS: A Candidate for an ARTEMIS Cross-Domain Reference Architecture for Embedded Systems*. 2009: Südwestdeutscher Verlag für Hochschulschriften (SVH) Aktiengesellschaft \& Co. KG.
169. Olivé, A. *Conceptual Schema-Centric Development: A Grand Challenge for Information Systems Research*. in *CAiSE*. 2005: Springer.
170. Omicini, A. and A. Ricci, *MAS Organization within a Coordination Infrastructure: Experiments in TuCSoN*. 2004.

171. Omicini, A. and F. Zambonelli, *Coordination for Internet Application Development.* Autonomous Agents and Multi-Agent Systems, 1999. **2**(3): p. 251--269.
172. Oreizy, P., N. Medvidovic, and R.N. Taylor, *Architecture-based runtime software evolution*, in *Proceedings of the 20th international conference on Software engineering*. 1998, IEEE Computer Society: Kyoto, Japan.
173. Oreizy, P. and R. Taylor, *On the Role of Software Architectures in Runtime System Reconfiguration*, in *Proceedings of the International Conference on Configurable Distributed Systems*. 1998, IEEE Computer Society.
174. Ottino, J.M., *Engineering complex systems.* Nature, 2004. **427**(6973): p. 399-399.
175. Pabjan, B., *Researching Prison--A Sociological Analysis of Social System.* Interdisciplinary Description of Complex Systems, 2005. **3**(2): p. 100--108.
176. Paolo, C. and R. Davide, *Jada - Coordination and Communication for Java Agents*, in *Selected Presentations and Invited Papers Second International Workshop on Mobile Object Systems - Towards the Programmable Internet*. 1997, Springer-Verlag.
177. Papadopoulos, G.A. and F. Arbab. *Coordination Models and Languages*. in *Advances in Computers*. 1998.
178. Papazoglou, M.P. and W.-J. Heuvel, *Service oriented architectures: approaches, technologies and research issues.* The VLDB Journal, 2007. **16**(3): p. 389--415.
179. Perry, D.E. and A.L. Wolf, *Foundations for the study of software architecture.* SIGSOFT Softw. Eng. Notes, 1992. **17**(4): p. 40-52.
180. Pezze, M., D. Tosi, and G.P. Picco. *Scavenging complex genomic information using mobile code: an evaluation*. in *Engineering of Complex Computer Systems, 2005. ICECCS 2005. Proceedings. 10th IEEE International Conference on*. 2005.
181. Picco, G.P., D. Balzarotti, and P. Costa. *LighTS: a lightweight, customizable tuple space supporting context-aware applications*. in *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*. 2005. New York, NY, USA: ACM.
182. Picco, G.P., A.L. Murphy, and G.C. Roman. *LIME: Linda meets mobility*. in *ICSE '99: Proceedings of the 21st international conference on Software engineering*. 1999. Los Alamitos, CA, USA: IEEE Computer Society Press.
183. Pierro, A.D., C. Hankin, and H. Wiklicky, *Probabilistic KLAIM*. 2004.
184. Pietzuch, P.R., *Hermes: A Scalable Event-Based Middleware*. 2004, Queens' College University of Cambridge.
185. Powell, D., *Group communication.* Commun. ACM, 1996. **39**(4): p. 50-53.
186. Pröstler, M., *Design and Implementation of MozartSpaces, the Java Reference Implemention of XVSM - Timeout Handling, Notifications and Aspects*. 2008, Institute of Computer Languages, Vienna University of Technology.
187. Rajendran, C. and O. Holthaus, *A comparative study of dispatching rules in dynamic flowshops and jobshops.* European Journal of Operational Research, 1999. **116**(1): p. 156-170.
188. Rakitin, S.R., *Manifesto Elicits Cynicism.* IEEE Computer, 2001. **34 (4)**.
189. Ratnasamy, S., P. Francis, M. Handley, R. Karp, and S. Schenker. *A scalable content-addressable network*. in *SIGCOMM'01: Proceedings of the 2001*

conference on Applications,technologies,architectures,and protocols for computer communications. 2001. New York,NY,USA: ACM Press.

190. Richards, M., R. Monson-Haefel, and a.D.A. Chappell, *Java Message Service*. Second Edition ed. 2009: O'Reilly.
191. Rowstron, A. and P. Druschel, *Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems.* Proc. of the 18th IFIP/ACM Int. Conf. on Distributed Systems Platforms (Middleware'01), 2001: p. 329--350.
192. Rowstron, A.I.T. and A.M. Wood, *BONITA: a set of tuple space primitives for distributed coordination.* System Sciences, 1997, Proceedings of the Thirtieth Hawaii International Conference on, 1997. **1**: p. 379-388 vol.1.
193. Roy Thomas, F., *Architectural styles and the design of network-based software architectures*. 2000, University of California, Irvine. p. 162.
194. Sancese, S., P. Ciancarini, and A. Messina, *Message Passing vs. Tuple Space Coordination in an Aerodynamics Application*, in *Proceedings of the 5th International Conference on Parallel Computing Technologies*. 1999, Springer-Verlag.
195. Satoh, F., Y. Nakamura, N.K. Mukhi, M. Tatsubori, and K. Ono. *Methodology and Tools for End-to-End SOA Security Configurations*. in *SERVICES '08: Proc. of the 2008 IEEE Congress on Services - Part I*. 2008.
196. Scheller, T., *Design and Implementation of XcoSpaces, the .Net Reference Implementation of XVSM - Core Architecture and Aspects*. 2008: Institute of Computer Languages, Vienna University of Technology.
197. Schreiber, C., *Design and Implementation of MozartSpaces, the Java Reference Implementation of XVSM - Custom Coordinators, Transactions and XML protocol*. 2008, Institute of Computer Languages, Vienna University of Technology.
198. Schwaber, K. and M. Beedle, *Agile Software Development with Scrum*. 2001: Prentice Hall.
199. Shannon, C.E., *A mathematical theory of communication.* ACM SIGMOBILE Mobile Computing and Communications Review, 2001. **5**(1): p. 3-55.
200. Shaw, M., *Comparing architectural design styles.* Software, IEEE, 1995. **12**(6): p. 27-41.
201. Shaw, M. and D. Garlan, *Software architecture: perspectives on an emerging discipline*. 1996: Prentice-Hall, Inc. 242.
202. Shull, F., V. Basili, B. Boehm, A.W. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz, *What We Have Learned About Fighting Defects*, in *Proceedings of the 8th International Symposium on Software Metrics*. 2002, IEEE Computer Society.
203. Siedersleben, J., *Moderne Softwarearchitektur*. 2004: dpunkt.verlag.
204. Slyngstad, O.P., J. Li, R. Conradi, and M.A. Babar. *Identifying and Understanding Architectural Risks in Software Evolution: An Empirical Study*. in *PROFES '08: Proceedings of the 9th international conference on Product-Focused Software Process Improvement*. 2008. Berlin, Heidelberg: Springer-Verlag.
205. Slyngstad, O.P.N., R. Conradi, M.A. Babar, V. Clerc, and H. van Vliet. *Risks and Risk Management in Software Architecture Evolution: An Industrial Survey*. in *Software Engineering Conference, 2008. APSEC '08. 15th Asia-Pacific*. 2008.

206. Solomon, S. and E. Shir, *Complexity; a science at 30.* Europhysics News, 2003. **34**(2): p. 54-57.
207. Stapleton, J., *DSDM: Business Focused Development*. 2003: Pearson Education.
208. Stephanos, A.-T. and S. Diomidis, *A survey of peer-to-peer content distribution technologies.* ACM Comput. Surv., 2004. **36**(4): p. 335-371.
209. Stephen, J.M., K. Scott, U. Axel, and W. Dirk, *MDA Distilled*. 2004: Addison Wesley Longman Publishing Co., Inc.
210. Stephens, M. and D. Rosenberg, *Extreme Programming Refactored: The Case Against XP*. 2003: Apress, Berkeley.
211. Sterritt, R. and M.G. Hinchey. *Biologically-inspired concepts for self-management of complexity*. in *Engineering of Complex Computer Systems, 2006. ICECCS 2006. 11th IEEE International Conference on*. 2006.
212. Stoica-Kluver, C. and J. Kluver, *Simulation of Traffic Regulation and Cognitive Developmental Processes: Coupling Cellular Automata with Artificial Neural Nets.* Complex Systems-Champaign-, 2007. **17**(1/2): p. 47.
213. Stoica, I., R. Morris, D. Liben-Nowell, D.R. Karger, M.F. Kaashoek, F. Dabek, and H. Balakrishnan, *Chord: a scalable peer-to-peer lookup protocol for internet applications.* IEEE/ACM Trans. Netw., 2003. **11**(1): p. 17â€'32.
214. Strom, R. and S. Yemini, *Optimistic recovery in distributed systems.* ACM Trans. Comput. Syst., 1985. **3**(3): p. 204--226.
215. Sussman, J., *Perspectives on Intelligent Transportation Systems (ITS)*. 2005: Springer, New York, NY.
216. Tanenbaum, A.S. and M.v. Steen, *Distributed Systems: Principles and Paradigms (2nd Edition)*. 2006: Prentice-Hall, Inc.
217. Taylor, R.N., N. Medvidovic, and E.M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*. 2009: Wiley.
218. Terry, W., *From programming environments to environments for designing.* Commun. ACM, 1995. **38**(6): p. 65-74.
219. Tolksdorf, R. and D. Glaubitz. *Coordinating Web-Based Systems with Documents in XMLSpaces*. in *CooplS '01: Proceedings of the 9th International Conference on Cooperative Information Systems*. 2001. London, UK: Springer-Verlag.
220. Tolksdorf, R., F. Liebsch, and D.M. Nguyen. *XMLSpaces.NET: An Extensible Tuplespace as XML Middleware*. in *In Report B 03-08, Free University Berlin, ftp://ftp.inf.fu-berlin.de/pub/reports/tr-b-0308.pdf, 2003. Open Research Questions in SOA 5-25 and Loose Coupling in Service Oriented Architectures*. 2004.
221. Triantafillou, P. and I. Aekaterinidis. *Content-based publish-subscribe over structured P2P networks*. in *International Conference on Distributed Event-Based Systems*. 2004.
222. van der Goot, R., J. Schaeffer, and G.V. Wilson. *Safer Tuple Spaces*. in *COORDINATION '97: Proceedings of the Second International Conference on Coordination Languages and Models*. 1997. London, UK: Springer-Verlag.
223. Vogel, O., I. Arnold, A. Chughtai, E. Ihler, T. Kehrer, U. Mehlig, and U. Zdun, *Software-Architektur: Grundlagen - Konzepte - Praxis*. 2008: Spektrum Akademischer Verlag.
224. Vrba, P. *MAST: manufacturing agent simulation tool*. 2003.

225. Vrba, R., V. Marik, and M. Merdan. *Physical Deployment of Agent-based Industrial Control Solutions: MAST Story*. in *IEEE International Conference on Distributed Human-Machine Systems*. 2008.
226. Walshe, R. *Modeling bacterial growth patterns in the presence of antibiotic*. in *Engineering of Complex Computer Systems, 2006. ICECCS 2006. 11th IEEE International Conference on*. 2006.
227. Wang, J., J. Cao, and J. Li. *Supporting Mobile Clients in Publish/Subscribe Systems*. in *ICDCSW '05: Proceedings of the First International Workshop on Mobility in Peer-to-Peer Systems (MPPS) (ICDCSW'05)*. 2005. Washington, DC, USA: IEEE Computer Society.
228. Weigand, H., F. van der Poll, and A. de Moor, *Coordination through Communication.* Proc. of the 8th International Working Conference on the Language-Action Perspective on Communication Modelling (LAP 2003), 2003: p. 1--2.
229. Wells, G., A. Chalmers, and P. Clayton. *Extending the Matching Facilities of Linda*. in *COORDINATION '02: Proceedings of the 5th International Conference on Coordination Models and Languages*. 2002. London, UK: Springer-Verlag.
230. Wells, G.C., *A Programmable Matching Engine for Application Develoment in Linda*. 2001, University of Bristol.
231. Wells, G.C., *Coordination Languages: Back to the Future with Linda.* Proceedings of the Second International Workshop on Coordination and Adaption Techniques for Software Entities (WCAT05), 2005: p. 87--98.
232. Wells, G.C., *New and improved: Linda in Java.* Sci. Comput. Program., 2006. **59**(1-2): p. 82--96.
233. Westerhoff, F.H., *Consumer Behavior and Fluctuations in Economic Activity.* Advances in Complex Systems, 2005. **8**: p. 209--215.
234. Woods, E. and N. Rozanski, *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives*. 2004: Addison-Wesley Longman.
235. Wooldridge, M., *An Introduction to MultiAgent Systems*. 2009, New York, NY, USA: John Wiley \& Sons, Inc.
236. Wyckoff, P., S.W. McLaughry, T.J. Lehman, and D.A. Ford, *T spaces.* IBM Systems Journal, 1998. **37**(3): p. 454--474.
237. Xu, P. and R. Deters, *Using event-streams for fault-management in MAS.* Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology, 2004 (IAT 2004), 2004.
238. Zaera, M., *WAVE-based Communication in Vehicle to Infrastructure Real-time Safety-related Traffic Telematics*. 2008, Telecommunication Engineering, University of Zaragoza.
239. Zavattaro, G., *Coordination Models and Languages: Semantics and Expressiveness.* Phd Thesis, Department of Computer Science, University of Bologna, 2000.
240. Zeidler, A., *A Distributed Publish/Subscribe Notification Service for Pervasive Environments*. 2004, Technischen UniversitÃ¤t Darmstadt.
241. Zloof, M.M., *Query-by-example: the invocation and definition of tables and forms*, in *Proceedings of the 1st International Conference on Very Large Data Bases*. 1975, ACM: Framingham, Massachusetts.