Separation and Adaptation of Concerns in a Shared Data Space

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Eindhoven, op gezag van de
Rector Magnificus, prof.dr.ir. C.J. van Duijn, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op dinsdag 27 juni 2006 om 16.00 uur

door

Giovanni Russello

geboren te Gela, Italië
Dit proefschrift is goedgekeurd door de promotoren:

prof.dr.ir. M.R. van Steen
en
prof.dr. M. Rem

Copromotor:
dr. M.R.V. Chaudron
Eerste promotor: prof.dr.ir. M.R. van Steen (Vrije Universiteit Amsterdam)

Tweede promotor: prof.dr. M. Rem (Technische Universiteit Eindhoven)

Copromotor: dr. M.R.V. Chaudron (Technische Universiteit Eindhoven)

Kerncommissie:
prof.dr. E.H.L. Aarts (Technische Universiteit Eindhoven)
prof.drs. M. Boasson (Universiteit van Amsterdam)
dr. A.M. Wood (University of York, U.K.)

The work in this thesis is supported by NWO as a part of project SACC (612.063.001).

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

IPA dissertation series 2006-11

© Giovanni Russello 2006. All rights are reserved. Reproduction in whole or in part is prohibited without the written consent of the copyright owner.

Printing: Eindhoven University Press

Cover design: Jan-Willem Luiten

Picture front cover: Dawn Breaking on Mt. Etna - Giovanni Russello
For more of Giovanni’s pictures see http://www.grussello.com
# Contents

0 Preface v

1 Introduction 1
   1.1 What it is all about ................................. 2
   1.2 Problem Description ................................. 3
   1.3 Contributions of This Thesis ........................ 4
   1.4 Organization of This Thesis ........................ 5

2 Separation by Coordination 7
   2.1 Introduction ....................................... 8
   2.2 Component-Based Approach ............................ 8
   2.3 Separation of Concerns ............................... 9
   2.4 The Shared Data Space Model ........................ 10
   2.5 Software Architecture =
      (Components + Coordination)/ SoC ..................... 14
   2.6 Separation by Coordination .......................... 14
   2.7 Related Work ...................................... 15
      2.7.1 Component-Based Approaches .................... 16
      2.7.2 Separation of Concerns ........................... 16
      2.7.3 Middleware .................................... 19

3 Performance 21
   3.1 Introduction ....................................... 22
   3.2 Data Space Strategies ............................... 23
      3.2.1 Strategy Evaluation ............................. 25
   3.3 Architectural Design for Performance ................. 26
      3.3.1 Operation Processing Subsystem ................. 27
      3.3.2 Adaptation Subsystem ............................ 29
      3.3.3 Autonomic Behavior of GSpace .................. 32
      3.3.4 Distribution Policy Evaluation .................. 35
      3.3.5 Automatic Profiling .............................. 38
      3.3.6 Adapting the Data Space Content ............... 39
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4</td>
<td>Implementation and Experiments</td>
<td>40</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Application Model Description</td>
<td>41</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Implementation Measurements</td>
<td>42</td>
</tr>
<tr>
<td>3.4.3</td>
<td>A Case for Dynamic Adaptation</td>
<td>45</td>
</tr>
<tr>
<td>3.4.4</td>
<td>Performance and Overhead</td>
<td>46</td>
</tr>
<tr>
<td>3.4.5</td>
<td>Accuracy of the Model</td>
<td>49</td>
</tr>
<tr>
<td>3.4.6</td>
<td>Choosing the Appropriate Threshold</td>
<td>51</td>
</tr>
<tr>
<td>3.5</td>
<td>Related Work</td>
<td>51</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Shared Data Spaces</td>
<td>51</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Adaptive Shared-object Systems</td>
<td>53</td>
</tr>
<tr>
<td>3.6</td>
<td>Conclusion</td>
<td>53</td>
</tr>
<tr>
<td>4</td>
<td>Availability</td>
<td>55</td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction</td>
<td>56</td>
</tr>
<tr>
<td>4.2</td>
<td>Architectural Design for Availability</td>
<td>57</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Autonomic Behavior in GSpace</td>
<td>58</td>
</tr>
<tr>
<td>4.3</td>
<td>Implementation and Experiments</td>
<td>64</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Setup of the Experiments</td>
<td>64</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Replication Policies</td>
<td>66</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Adding Awareness of Node Availability to Policies</td>
<td>67</td>
</tr>
<tr>
<td>4.3.4</td>
<td>Dealing with a Changing Environment</td>
<td>70</td>
</tr>
<tr>
<td>4.3.5</td>
<td>Adaptation Mechanism Performance</td>
<td>73</td>
</tr>
<tr>
<td>4.4</td>
<td>Related Work</td>
<td>80</td>
</tr>
<tr>
<td>4.5</td>
<td>Conclusions</td>
<td>82</td>
</tr>
<tr>
<td>5</td>
<td>Security</td>
<td>83</td>
</tr>
<tr>
<td>5.1</td>
<td>Introduction</td>
<td>84</td>
</tr>
<tr>
<td>5.2</td>
<td>Security Policies and Threats</td>
<td>85</td>
</tr>
<tr>
<td>5.3</td>
<td>Architectural Design for Security</td>
<td>87</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Threat Model</td>
<td>88</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Security Modules</td>
<td>88</td>
</tr>
<tr>
<td>5.3.3</td>
<td>An Example of a Security Policy</td>
<td>91</td>
</tr>
<tr>
<td>5.4</td>
<td>Managing Security Properties</td>
<td>101</td>
</tr>
<tr>
<td>5.5</td>
<td>Personal Health Systems–The Philips Case Study</td>
<td>102</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Motivations and Assumptions</td>
<td>103</td>
</tr>
<tr>
<td>5.5.2</td>
<td>User Scenarios</td>
<td>105</td>
</tr>
<tr>
<td>5.5.3</td>
<td>Deployment of a PHS Based on GSpace</td>
<td>106</td>
</tr>
<tr>
<td>5.5.4</td>
<td>Collecting Data from Devices</td>
<td>110</td>
</tr>
<tr>
<td>5.6</td>
<td>Related Work</td>
<td>116</td>
</tr>
<tr>
<td>5.7</td>
<td>Conclusions</td>
<td>118</td>
</tr>
<tr>
<td>6</td>
<td>Combining Concerns</td>
<td>121</td>
</tr>
<tr>
<td>6.1</td>
<td>Introduction</td>
<td>122</td>
</tr>
<tr>
<td>6.2</td>
<td>Requirements for a Multi-Concern Architecture</td>
<td>122</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>6.3</td>
<td>Concern Interactions</td>
<td>124</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Availability and Performance Interactions</td>
<td>124</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Availability and Security Interactions</td>
<td>127</td>
</tr>
<tr>
<td>6.4</td>
<td>Design for a Multi-Concern Architecture</td>
<td>128</td>
</tr>
<tr>
<td>6.5</td>
<td>Summary</td>
<td>130</td>
</tr>
<tr>
<td>7</td>
<td>Conclusions</td>
<td>131</td>
</tr>
<tr>
<td></td>
<td>References</td>
<td>135</td>
</tr>
<tr>
<td>A</td>
<td>Alternative Strategy for Adaptation</td>
<td>155</td>
</tr>
<tr>
<td>A.1</td>
<td>Introduction</td>
<td>156</td>
</tr>
<tr>
<td>A.2</td>
<td>Modifications</td>
<td>156</td>
</tr>
<tr>
<td>A.3</td>
<td>Notation</td>
<td>157</td>
</tr>
</tbody>
</table>
Chapter 0

Preface
July 2000, a hot summer afternoon in Catania. After several hours of talking about CORBA and Obliq, Vladimiro asked: “So, what about becoming a Ph.D., uh?” “What’s a Ph.D.?” replied Giovanni.

Vladimiro was my thesis supervisor at that time. His sudden suggestion of me doing a Ph.D. sounded so exciting that I instantly flushed out my previous plans of going to work for a company in Catania. When asked about the reasons for becoming a Ph.D. student, my thoughts race back to that afternoon.

How I ended up in The Netherlands and have spent the next five years there accomplishing this manuscript is another story. I can surely say that Michel and Maarten have been the main plotters behind the scene of all of this.

I would like to use this page to thank all the people that have been around me during this period of my life. It is my intention to avoid to turn this page in a list of names, looking like one from the Old Testament. Not to mention the risk of forgetting someone’s name. Thus, I prefer to multicast a “Thank You” to all of you that have been around me during this five years. For, in one way or another, I have involved you in the writing of these pages. During discussions in the office, coffee breaks, volleyball and football matches, heavy training sessions in the gym, lunches and dinners, my colleagues and friends have always been supportive and helpful in finding solutions to my (endless) problems. I leave up to the reader to decide whether she or he falls in one of these categories. My hope is that you will enjoy reading the rest of this thesis as much as I enjoyed writing it.

Concluding, I would like to thank the pillars of my life, my Parents. Without their constant support I could not be where I am now.

May 24, 2006.
On a train to Nottingham.
Chapter 1

Introduction
1.1 What it is all about

This thesis is about distributed systems. The main characteristic of a distributed system is its capability of exchanging messages among physically separated entities. To achieve this, distributed systems rely on networks.

Distributed systems are realized for fulfilling several goals. Distributed systems make possible remote access of resources, both hardware and software. Moreover, they make possible to share a common workspace establishing a collaborative environment where multiple users can participate. Typically, distributed systems hide for the user the fact that resources and processes are distributed across several computers. By hiding such details, distributed systems provide a high level of abstraction that makes them easier to use. Another advantage of distributed systems is scalability. If well designed, a distributed system scales better than a centralized system, although at the cost of more resource usage.

However, there are some issues that need to be considered when designing a distributed system. In particular, in this thesis, we will concentrate on the following issues:

- **How to distribute data for performance**: data needs to be made available to processes and users. Ever since distributed systems appeared, different strategies for distributing data have been devised. These strategies try to strike a balance between lowering the latency required for accessing the data and the resources used for distributing the data. For instance, replication is one strategy that plays a major role in distributed systems. By replicating data, it is possible to make copies of the data available closer to the location where it is needed, thus lowering access time. However, this comes at costs in terms of bandwidth and memory usage for keeping such data consistent. Thus it is not always good practice applying replication. The issue here is to find which specific distribution strategy should be used for the data in a distributed systems. As we will see, the right decision is dictated by several factors that the system designer cannot completely know at design time. Among those factors we mention application usage patterns and network characteristics.

- **How to support fault tolerance for data**: in a distributed system, nodes may fail or the network can be partitioned. As a result, the data that is stored in certain nodes may became unavailable. A common solution to this problem is that of replicating data. By replicating data in several nodes, the system makes sure that data is statistically available even if some of the nodes are down. Several ways of replicating data are possible. Some of them enforce a strict model of consistency to all copies. Others use a more relaxed model. This means that the resource usage of each solution is different from the others. The main problem resides in finding which replication strategy
1.2 Problem Description

Over the last 10 years, distributed systems have developed from interesting laboratory curiosities to commercially deployed applications. We constantly experience interaction with one or another type of distributed system: the Internet, online banking, and telecommunication networks are just a few examples of distributed applications that are commonly used every day by millions of people.

Each distributed application has specific requirements. For instance, online banking requires high availability and security while telecommunication applications require high availability, throughput, and performance. A common way to fulfill these requirements is by the use of a *middleware*. Middleware is dedicated software that provides services for enforcing system-wide properties, but which is independent of applications.

*should be applied to the data in the system.* Also for this case, the system designer should be able to foresee which usage pattern the application that uses his system has. Moreover, he should know the availability characteristics of the nodes where the entire system is deployed.

- **How to enforce data security:** security is one of the most important aspects of a distributed system. In a distributed system, security mainly deals with two aspects. One aspect deals with the enforcement of privacy in the communication between the different entities that form the systems. The other aspect concerns the authorization rights that entities have on data. There are several mechanisms that can be used for both aspects. Placing all security mechanisms into a system is not a sound idea. What is important is to know what the threats are that the system should be protected from and then use the right mechanisms for doing the job. For its nature, security needs to be pervasive throughout the entire system. Often, this means that once a specific mechanism is used in a system the mechanism becomes part of the system. Modifications to this mechanism requires an invasive change in the system as all. The problem that arises is how to deal with security matters in an environment where security requirements change over time.

In this thesis we will touch upon each of the aforementioned issues. The main focus of this thesis is **flexibility**. We want to design a framework that helps the system architects in designing their distributed systems in a way that decisions taken at design time could be easily changed after deployment time.

More importantly, the approach that is discussed in this thesis does not just apply for the current state of solutions for distributed systems. Our approach is **extendable**, in the sense that even in the future when new problems arise and new solutions are provided these solutions can be incorporated in our approach.
By using middleware, the application developer is relieved from implementing those services himself. However, once the developer chooses a middleware then his application is committed to the specific middleware’s mechanisms.

As discussed in [67], most of the modern middleware systems do not support a clear separation between the basic functionality of an application from the system-wide properties that should be enforced. This leads to an increase in the complexity of the application, as pointed out by Filman, et al. in [27]. This practice negatively affects the system as a whole because of:

- poor readability of the code: the application code is scattered with code fragments that do not deal with the application functionality. A developer reading the code is expected to be also an expert for the other concerns.
- poor maintainability of the code: debugging and improving application code becomes more cumbersome since details about each concern need to be taken into account.
- reduced reusability: if the application code is deployed in another environment with different properties then the code needs to be changed to be adapted to the new environment.

Our goal is to design a architecture where application design and specification are done in a modularized fashion: application functionality and system-wide concerns are isolated from each other with the benefit of facilitating development, reusability, and maintainability of each concern.

1.3 Contributions of This Thesis

Our architecture is based on the shared data space model. The extensions that we have introduced to the model for an extensive support of several extra-functional concerns are transparent to the application. As a matter of fact, in the simplicity of the interface of the shared data space resides its strength for realizing distributed applications.

The main reason for choosing the shared data space model comes from the fact that this model supports naturally the separation of the computational part of an application from its coordination part.

The contributions of this thesis can be summarized as follows:

- Propose a specific architecture for separating application functionality from extra-functional concerns. Each extra-functional concern is catered by a set of mechanisms that is available in the framework. New mechanisms can be deployed, extending the life cycle of the framework.
• Our architecture supports modularization of concern specifications. Functional and extra-functional concerns can be specified in complete isolation to each other. This allows specialists to work on mechanisms specific for a concern without having to know too many details of other concerns.

• As for the shared data space model, we claim that our architecture is the first in supporting the separation of policies that drive the system from the mechanisms that are employed. This increases the flexibility of the architecture, since by using the appropriate mechanisms the architecture can be deployed in different environments.

• Another contribution of this thesis is the modularization of the coordination model. Typically, such model is used for abstracting from the underlying infrastructure where applications are deployed. With our approach, we can address specific concerns, such as performance, availability and security. As a consequence, such concerns are encapsulated in well-defined modules enhancing the reusability of software artifacts. Not only application code can be reused, but also policies and mechanisms represent reusable units. System architects need not to re-invent mechanisms when the system is deployed in other similar environments.

• By separating application functionality from extra-functional concerns, it becomes possible to provide a dynamic adaptation of the mechanisms that cater for those concerns during run-time. The architecture offers a feedback loop that monitors the application’s behavior. Whenever the application’s behavior changes the feedback loop adapts the mechanisms to the current needs of the application. This is another unique feature of our architecture among other shared data space systems.

1.4 Organization of This Thesis

Chapter 2 introduces the reader with the main concepts of this work. Here is where we discuss the origin of the shared data space and the principle of separation of concerns. Moreover, we provide a overview of other methods related to our approach.

Chapters 3 to 5 focus on designing, implementing, and testing architectures to deal with performance, availability and security concerns, respectively. In Chapter 6 we provide some thoughts on how a multi-concern architecture can be conceived where those concerns could be dealt with at the same time. We conclude in Chapter 7.
Chapter 2

Separation by Coordination
2.1 Introduction

The general aim of our research is to provide support for the development and deployment of distributed component-based applications. What often makes these applications so complex is that they need to exhibit specific behavior that is inherently strongly dependent on external, and often unstable factors, such as resource availability. As a consequence, much effort is put into developing program segments by which the effects of these external factors become more controlled. Typical examples include program fragments for data availability, fragments for data distribution to meet performance demands, and fragments for enforcing security policies.

In this chapter, we will present our framework that facilitates the development and deployment of distributed component-based applications. We will introduce the technologies and concepts that are the foundation of this thesis. We will show how the combination of their use allows us to build our framework in a systematic and principled manner.

The structure of this chapter is as follows. In Section 2.2 we describe the Component-based approach that we employ for building applications. In Section 2.3 we introduce our ideas on how the separation of concerns principle can be applied to the design and implementation of distributed systems. Section 2.4 introduces the basic concepts behind the shared data space model. Section 2.5 provides an overview of the project from which the work described in this thesis was generated. In Section 2.6 we explain the role that coordination plays in this approach. We conclude this chapter by discussing some related work in Section 2.7.

2.2 Component-Based Approach

The component-based approach is one of the most recent concepts introduced in software engineering [84]. A software component is a unit of deployment that encapsulates functionality. A component is an autonomous entity that implements one or more interfaces. The interface(s) that a component implements represents the contract between the component and the clients (often other components) that use it.

Components can be developed and tested independently. Applications can be created by composing off-the-shelf components. By reusing existing components, building an application is faster and less prone to errors or disfunctions.

Methods in which application can be built by composition of components can be classified as: static and dynamic composition.

Static composition refers to a method where components are glued together before run-time. Usually, a compiler is used. In this method, changing or adding a com-
ponent requires halting the application to re-bind the components together. On the other hand, applications built using this method show a higher computational speed since communication between components is handled as a direct method (or function) invocation.

Dynamic composition refers to a method where components can be bound at runtime and changing or adding a component does not require halting the application. This method is particularly indicated for those applications with high availability requirements. This method relies on a software layer that glues components together. This software layer goes with the name of middleware. The middleware provides some services to components that can be invoked by means of an interface. Therefore, applications built using this method are bound to middleware-specific interface. Any new component that is added to the application has to implement or use the middleware interface. Moreover, because components communication are mediated by the middleware, applications composed by this method can have a lower computational speed.

In the framework that will be discussed later on in this chapter, we employ dynamic composition as method for building application. As discussed in Section 2.4, we employ the shared data space model as compositional glue for binding components together. In the following section, we discuss the general principle that drives the design of our framework.

2.3 Separation of Concerns

Software engineering promotes the improvement of quality of software artifacts, the reduction of software production costs, and facilitates software evolution and maintainability. Research in the software engineering domain has proposed approaches that to some extend fulfil part of the promised goals. According to [86], the complete fulfillment of those goals is jeopardized by the fact that the proposed approaches neglect a full realization of Separation of Concern (SoC).

According to SoC, all relevant concerns of a software system should be treated as separated modules. Current approaches provide separation to a certain level, together with decomposition and composition mechanisms. However, these approaches provide a single dimension of separation, with a limited set of tools for decomposition and composition.

Let us take as an example of a distributed application developed using an object-oriented approach. The application is decomposed into classes. Classes can be further decomposed into interfaces and units of implementations. These modules should be used for defining the basic functionality of the applications. However, since our application is distributed there are other concerns that need to be taken into account. For instance, how should the data be distributed, and who and how should the data be accessed, are typical concerns that are not directly involved
with the basic functionality of an application. We refer to these concerns as extra-functional concerns.

The object-oriented approach does not offer any specific modules for dealing with extra-functional concerns. As a result, code fragments concerning data distribution and security become entangled with code concerning application functionality and can result in code that is scattered across several objects.

As a consequence, the following properties of applications are negatively affected:

- Application evolution: because the units of abstraction do not match with the different concerns, changes in one concern result in complex and invasive changes in all modules.

- Reusability of concerns: as the code dealing with the application logic can be reused, the same should hold for code that deals with extra-functional concerns, such as performance and security. However, reusing a security mechanism in a new context proves to be difficult because the code for the mechanism has been merged with application code.

- Concern traceability: traceability among different concerns is limited. This is mainly due to the imposed decomposition in objects that blurs the boundaries between each concern.

The above problems would be ameliorated if software were built in a fully modularized way where different concerns are separated from each other. As discussed in [27], the SoC principle is essential for developing applications that strongly depend on their environment or that can be expected to continuously evolve in the course of time.

However, concerns are not completely orthogonal to each other. They overlap influencing one another at run-time. An effective framework for SoC should provide flexibility to support all concerns simultaneously, and be robust to manage overlaps and interactions among concerns.

### 2.4 The Shared Data Space Model

The shared data space model was introduced by the coordination language Linda [29]. Storage in Linda takes place in a so-called tuple space. In this tuple space, data is stored as persistent objects, called tuples.

Linda provides three basic operations: `out`, `in` and `rd`, and two variant forms, `inp` and `rdp`. The `out` operation inserts a tuple into the tuple space. The `in` and `rd` operations respectively take (destructive) and read (non-destructive) a tuple from the tuple space, using a template for matching. The tuple returned must exactly match
every value of the template. Templates may contain wildcards, which match any value. Whereas putting a tuple inside the tuple space is non-blocking (i.e., the process that puts the tuple returns immediately from the call to `out`), reading and taking from the tuple space is blocking: the call returns only when a matching tuple is found. The `inp` and `rdp` are predicate versions of `in` and `rd`: they too try to return a matching tuple. However, if there is no such tuple they do not block but return a value indicating failure. In our current implementation of a SDS the `inp` and `rdp` operations are not supported. However, the behavior of those operations can be mimicked by the `in` and `rd` operations with the use of a `timeout` parameter. The timeout parameter specifies the time that the operation is blocked waiting for a matching tuple to be inserted in the space. Setting the timeout parameter to 0 signifies that the operation has not to be blocked if no matching tuples are in the space.

In Linda it is also possible to fork a process inside a tuple space through so-called live tuples. To insert a live tuple inside a tuple space the `eval` operation is used. `eval` is similar to an `out` and it is specific for live tuples. Once a live tuple is inserted in a tuple space it carries out the specified computation. Afterwards, a live tuple turns into an ordinary data tuple, and it can be used as such.

The shared data space model was developed as a paradigm for coordinating parallel and distributed applications. It falls in the category of middleware software. Coordination through shared data spaces, also called generative communication, forms an attractive model for developing distributed and component-oriented systems as it supports referential and temporal decoupling of processes [30]. Referential decoupling means that components exchange data without the need of knowing each other. Temporal decoupling means that those components do not even have to be online at the same time. This way, components can be connected to or disconnected from the data space at any time, making them easier to combine or replace.

Coordination systems have also been successfully adopted in developing commercial applications. In particular, SPLICE [11] has been used in the development of command-and-control, and traffic management applications. As reported in [10], the use of the shared data space model has been the key factor in reducing the complexity of developing such applications.

Retrieving a tuple generally requires an associative searching technique. As it turns out, general-purpose distributed associative searching is difficult to implement. For this reason, many systems implement a tuple space by means of a single server, as shown in Figure 2.1(a).

Problems of this centralized approach are performance and scalability when components are placed far apart, or when (part of) the network offers only limited quality of service. A solution to these problems is to partition and replicate the tuple space across multiple servers, as shown in Figure 2.1(b).
This approach has been widely applied in the parallel programming community where efficient implementations of the Linda tuple space have been sought by statically distributing tuples using compile-time analysis. Unfortunately, providing efficient implementations of the shared data space model shows to be difficult. This is especially so when dealing with large-scale systems in which components can be numerous and placed far apart (see also [68]). Matters are further complicated when, for example, timing constraints need to be taken into account. Attaining efficiency is difficult mainly due to the combination of distribution and the associative nature of the generative communication model. Generative communication essentially introduces a search through a data space in order to match data to some given template. In a distributed system, such a search will generally involve searching through data that is spread across multiple machines.

When dealing with highly dynamic environments, where application components unpredictably join and leave the system, we argue that better results can be achieved if coordination is combined with separation of concern techniques. In particular, we believe that by separately specifying extra-functional concerns for tuples it is possible to derive an efficient implementation of the coordination between components. The design of such a framework is the main topic of the following section.

**Figure 2.1** (a) Configuration with distributed applications and centralized tuple space, and (b) one with a distributed tuple space.
2.4 The Shared Data Space Model

Figure 2.2 Our approach for an architecture that supports Separation of Concerns.


Chapter 2 Separation by Coordination

2.5 Software Architecture = (Components + Coordination)/ SoC

The work done in this thesis is part of the SACC (Software Architecture = Components + Coordination) project. The main goal of the project is to provide a framework that can reduce the effort spent in designing distributed applications. The project has two main research tracks: theoretical and practical.

The theoretical track focuses on generating a foundation that supports provably correct methods for getting from the specification to an implementation of an application. Here, the definition of the basic functionality of an application is done by using an abstract language called GAMMA [6]. In GAMMA a program is seen as a the execution of atomic actions called rules. Rules operate by rewriting the content of a multiset in a shared data space. A rule is executed as long as the data in the multiset can fire that rule. When the data in the multiset cannot fire any rules then the program terminates. This is somewhat similar to a chemical reaction. For this reason, the GAMMA model is referred to as the chemical reaction model. The extra-functional concerns are specified using tailor-made design languages. The definition of such languages together with their operational semantics can be found in [56].

The practical track focuses on designing a suitable framework for the theoretical models. The design and implementation of such a framework, together with its experimental evaluations is the main focus of this thesis.

2.6 Separation by Coordination

Fundamental to coordination-based systems is the separation between a computation model and a coordination model. The computation model captures application functionality; the coordination model links the various application activities by providing facilities for communication and synchronization. This separation is a prerequisite for any model that aims to support separation of concerns.

Instead of seeking universal solutions, we argue that the middleware should provide the mechanisms by which various application-specific solutions can easily be supported. The novelty of our approach lies in the way that we separate in our architecture the modules that address extra-functional concerns from the basic functionality of the application.

We argue that for distributed component-based applications, separating concerns should be supported by the middleware on which the applications are executed. Interweaving of policies should take place at run-time such that it becomes possible to change policies without the need for recompiling the application source code.
Figure 2.2 depicts the approach that we propose for system design and implementation. Each application is specified in terms of its functionality along with a number of extra-functional concerns. Functional specifications are transformed into components that are subsequently mapped onto the various nodes that make up the distributed system. Extra-functional concerns are transformed into policy descriptors that are downloaded into the middleware where they are interpreted at run-time by various concern managers.

Figure 2.2 presents the case in which two applications, A and B, are specified and deployed. We use grey blocks for application A and dashed line blocks for application B. For both applications, functional specifications are transformed to a set of components. For A this leads to components AC1 and AC2, while component AC3 belongs to B. Extra-functional concern descriptions for either application are downloaded as policy descriptors into the application-specific layer of the middleware. Each application can thus define its own policies independent from other applications and from the system-wide policies. The figure also shows the integration with system-wide policy descriptors, such as those used for global scheduling or security. These descriptors are downloaded into the middleware at the level of the system layer.

Besides having application-specific policy descriptors, it is desirable to support the specification of global, system-wide aspects. These aspects are transformed into application-independent policy descriptors.

Distributed component-based applications may be unpredictably dynamic. Components may join and leave. Due to connectivity, nodes can appear and disappear without any warning. This can make a tailored system obsolete after a while since the application was deployed. The design of our architecture is such that the hooks where the extra-functional concerns are injected in the system allow the changing of a specific concern descriptor at run-time without the need to stop the application. We exploit this feature of our architecture for dynamically adapting its internal structure to overcome the application changes during run-time. For this reason, we inserted in the middleware the Feedback/Adaptation Subsystem. This layer provides a feedback loop. The execution of the loop consists of monitoring the environment where the system is deployed and quantifying how the system is performing. If the adaptation subsystem finds out that a better configuration can be achieved then it can autonomously adapt the internal structure of the middleware to the current situation.

2.7 Related Work

The realization of our approach is based on the confluence of three key technologies: component-based design, separation of concerns, and middleware. In this section, we discuss other works that are related to our research. We will categorize them
according to the type of technology that makes them relevant to our approach.

2.7.1 Component-Based Approaches

The component based approach was introduced for the first by McIlroy [49], in 1969. However, only in the last decade has this approach become a viable alternative to object-oriented programming. This is confirmed by the number of commercial products that are available, such as Microsoft Component Object Model (COM) [52], Microsoft .NET [54], and Sun Enterprise JavaBeans (EJB) [82].

Among those, the EJB approach is closest to ours. The EJB model allows the application designer to concentrate on the application functionality without having to take into account other concerns, such as security and transactional behaviors. EJB provides a component-based approach for designing and deploying server-side enterprise applications. Beans take care of security and transaction integrity and assembled together by the assembly tool.

2.7.2 Separation of Concerns

Currently, the most active approach in separation of concern is that of Aspect-Oriented Programming (AOP) [1, 42, 91]. In AOP, aspects (terminology used in AOP for extra-functional, possibly crosscutting, concerns) such as quality of service, fault tolerance, energy consumption, and security, are captured as aspects and are separated from other parts of applications. Aspects are specified during the developing phase and woven with the application code during compile time (sometime at run-time). A compiler, called aspect weaver, is responsible for the weaving procedure. The locations in the program where aspect code is inserted are called pointcuts and often they are identified in the application at designing time.

Depending on how the weaving process is carried out, AOP approaches can be categorized in two groups: static and dynamic. In the following, the most prominent works of each of the two groups are discussed.

Static AOP In these approaches, aspects are woven at compile time, producing a tangled executable that cannot be dynamically reconfigured. In the target code, the separation between aspects and application functionality is lost. One of the most vivid example in this group is AspectJ [43]. AspectJ extends the Java programming language introducing specific modules, called aspects, where crosscutting are specified. Aspects are woven into the application code during compile time, using the AspectJ compiler. The AspectJ compiler uses the joint points inserted in the application code for weaving the aspect code. Some of the join points that AspectJ support are the following: method calls, constructor calls, constructor executions, field references, field sets, error handler executions. The
weaving process produces Java bytecode that can loaded in a standard Java Virtual Machine. It is important to note that AspectJ allows the specification of crosscutting concerns as aspects as long as these concerns cross-cut the join points that are defined in AspectJ.

OpenJava [85] is another extension of the Java programming language. More precisely, OpenJava extends the reflection mechanism of Java. By using a source-to-source compiler, Java applications are extended with behavioral reflection facilities that can manipulate program entities at run-time.

The FRIENDS system [25] is a framework for developing distributed applications that provides a set of libraries for dealing with fault tolerance, secure communication, and group communication. The application developer selects which of these libraries must be used in his application. A dedicated compiler entangles these libraries with the target code of the application. Thus, at run-time, the appropriate libraries are executed. However, since these libraries are part of the executable code they cannot be changed at run-time. This limitation of FRIENDS is overcome in the Programming Control Logic (PCL) [2]. PCL provides a framework where libraries for aspects are compiled and linked with the application code. Further, in PCL the executable code is provided with multiple libraries, thus allowing changing of libraries at run-time. The downside here is that the set of interchangeable libraries is limited to those libraries that have been compiled with the application code before run-time.

**Dynamic AOP** Unlike static AOP, in dynamic AOP aspects are fully separated from the application code.

Adaptive Java [41] extends Java introducing new language constructs that support behavioral reflection. These extensions offer to the application developer means for monitoring and adapting the behavior of the application during runtime. The aspect code is wrapped around the application code. The aspect code and application code communicate via a well defined interface. The use of the wrapper-pattern guarantees that aspect code is not entangled to the application code and thus aspect code can be dynamically changed at run-time. However, this technique requires the application developer to modify the application code to accommodate the interaction with the aspect code.

In Composition Filters [7] crosscutting concerns are disentangled from the application code by means of *filters*. These filters intercept the messages that objects exchange (either locally or remotely) during execution. Filters massage and check the messages before they are delivered to objects. In this way, filters can be used for aspects such as security authentication and encryption. Filters are engaged by means of a well-defined wrapping interface that is added to objects in a pre-compile phase. However, filter code is maintained separated from the application code. This allows insertion and modification of filters at run-time without recompiling the application.
Other approaches use a two-step mechanism to support run-time adaptation of crosscutting concerns. In the first step, generic interception hooks are woven into the application code using some existing aspect compiler (such as the AspectJ compiler). These hooks are used in the second step to intercept the program flow. When a hook intercepts an operation it activates the appropriate aspect code. Note that, because aspect code in not entangled with the application code it is possible to dynamically change aspect code. ARCAD [22] and TRAP/J [73] fall in this category. The only difference between these two approaches resides in the granularity of the interception hooks. In ARCAD hooks can intercept only the message flow between objects. In TRAP/J hooks have a finer granularity because they can also intercept method invocations.

Another way of supporting dynamic AOP is to weave application code with the aspect code at load time. Approaches such as JOIE [18], JBoss [36], Kava [93], and R-Java [32] use modified class loaders that insert traps in the class files (bytecode) to redirect the execution to appropriate aspect code. Again in this approach, the aspect code is not part of the application code and can be dynamically changed at run-time.

**Discussion**  Common to the AOP approach is the extension of an existing programming language with some aspect-oriented constructs to isolate extra-functional concerns. This not only requires the expansion of the target programming language with new keywords, but more importantly it requires either to modify standard tools (such as compilers and class loaders) or to introduce dedicated ones. Therefore, the reuse of code (both of applications and aspects) is constrained by the specific tool that was used to weave the application and aspect code. This is not the case of our approach. In our framework, the code for the application functionality and for the extra-functional concerns are not bound to any programming language or tool. It is the framework that mediates during the execution of operations (on behalf of application components) interactions between application and extra-functional concern. The joinpoints, to use an AOP terminology, are in the architecture of our framework. Interaction between application and kernel is based on an inter-process protocol over a socket layer. Therefore, it is not required that application and framework are implemented using the same programming language. The only dependency to a specific programming language can be found in the execution of the extra-functional code in the framework. However, this limitation can be overcome using source-to-source compiler to translate code from one language to the target code of the framework.

Another important aspect of AOP approaches is that aspects can be woven into the application code only where pointcuts (or any hook used for intercepting the execution flow) can capture the execution flow of programs. In its current state, AOP approaches are designed to deal with the object-oriented paradigm. This means that AOP is not transparent to the paradigm used by a specific application. In our approach no assumption is made on the application paradigm. This has the clear benefit of increasing the reusability of code for extra-functional concerns.
2.7.3 Middleware

Middleware is defined as a generic software layer that separates applications from platform-specific details, such as operating systems and network protocols. This allows the same applications to run on heterogeneous environments and to interoperate with diverse applications. Middleware provides a high-level of programming abstraction, such as remote objects, making the programming of distributed applications similar to stand-alone programs. Applications can use the services that the middleware implements. These services can be generic, such as fault tolerance, security, transaction mechanism, and they can be specific services tailored to particular types of applications. Notable works in this area are Common Object Request Broker Architecture (CORBA) [57], Java Remote Method Invocation (RMI) [80], and Distributed COM (DCOM) [53].

Recently, a lot of research has been aimed at on providing adaptability at the level of middleware to support to some extent dynamic composition of crosscutting concerns. These projects can be categorized in the following groups.

In the first group there are approaches that provide a layer of adaptable communication services, such as transactions, security authentication, security communication, quality of communication, etc. Mainly, these approaches work by intercepting messages between remote objects and triggering the appropriate implementation of a service. Most of these approaches are extension of CORBA. A representative sample of these approaches is listed as follows: ACE [76], TAO [75], CIAO [92], DynamicTao [44], Open ORB [9], OpenCorba [45], Orbix/E [34] and Eternal [55]. Some approaches in this category require that the application code is modified to explicitly incorporate calls to services provided by the middleware (e.g., OSG [26] and QuO [96]).

Other approaches provide extensions of standard platform services, such as extensions of socket implementations (e.g., MetaSockets [74], Rocks [95], and Racks [95]). These approaches extend the capabilities of standard sockets by using filters to intercept packets that applications send via a socket. Filters are used to encrypt/decrypt packets, or to resume session automatically after a disconnection.

Yet another group of approaches are based on modifications of the virtual machine. These approaches intercept local and remote messages that go through the virtual machine. Interceptions trigger the code for handling the specific concern and that can be adapted to the specific circumstances. Approaches in this group may extend a standard virtual machine (e.g., PROSE [63, 64], Guaraná [58], and Iguana/J [66]) or may customize a version of the standard virtual machine (e.g., Java Micro Edition [83]).

Most of these approaches are focused to the object-oriented paradigm. As a matter of fact, most of the projects mentioned above are extensions to CORBA. Distributed object-oriented applications are notoriously difficult to manage in a dy-
namic environment since they tend to establish a synchronous type of communication. In our framework, the shared data space is used for mediating communication among application components. Clearly, the main advantage of our approach is the uncoupling of the artifacts that compose the application. Consequently, applications can easily be (re)composed dynamically during execution time. As for the rest of the approaches presented above (namely those that modify the transport layer (sockets) and the virtual machine) they fail in that the level of abstraction that they provide is not abstract enough to support an agile development of applications. Our main contribution in this respect is that we extended the shared data space idea with separation and adaptation of extra-functional concerns that further extends the set of properties of the shared data space model.

Shared data space implementations also fall in the middleware category. There has been a lot of interest in this area leading to a plethora of different approaches. Some of those approaches are presented in the following chapters, classified according to the type of concerns that these implementations address. Important for its relevance to our approach is the SPLICE architecture. The design of the SPLICE architecture was driven by the application domain where it has been employed, that is large-scale distributed embedded applications (e.g., traffic control, command-and-control, process control). This type of applications is very difficult to design and deploy in that they exhibit a very dynamic behavior and have very strict requirements in terms of timeliness, robustness, reliability and maintainability. The SPLICE designers recognized that to alleviate the design of such applications functionality concern should be separated from those extra-functional concerns. The shared data space model naturally supports this separation. As such, the SPLICE architecture is actually an implementation of a SDS; it is worth noting that the design of SPLICE started in the late 70s. In SPLICE, application functionality is mapped into application components and separated from the extra-functional concerns that are either embedded in the middleware or supported by it. The SPLICE architecture explicitly addresses the following extra-functional concerns: data distribution, data temporal aspects, fault-tolerance, and application maintainability. SPLICE is related to our approach in that there is a clear attempt to address extra-functional concerns separately form application functionality. However, in SPLICE certain concerns are hard-coded in the architecture and tailored to the specific application domain (such as low latency), while others are supported by the architecture and require application components for their implementation. In our approach, we support separation of concerns in a more orthogonal way. In fact, mechanisms used for dealing with extra-functional concerns are not hard-coded in the architecture but treated as separated modules that can be adapted to the specific needs of the application.
Chapter 3

Performance
3.1 Introduction

Software engineering artifacts are witnessing increasing demands for interconnectivity, adaptivity, and flexibility. Existing systems need to exchange information, even in the presence of transient connections; they need to adapt dynamically to different usage contexts; and their structure should support the addition and removal of functionality.

This leads to architectures for distributed component-based systems (DCBSes) where components may dynamically join and leave the system at run-time. The dynamic evolution of the configuration of applications poses new challenges to the balancing between resource usage and performance optimization.

Dynamic composition and reconfigurations of DCBS applications is achieved by means of a software layer, called middleware. The middleware has to provide the infrastructure that allows components to communicate. Since components are subject to reconfigurations it is important that they are loosely coupled. This decoupling can be realized in two dimensions: time and space.

Generative communication [29], also referred to as data-oriented coordination [60], provides both types of decoupling. In the literature several implementations of generative communication using shared data space systems exist.

A drawback of those systems is their use of a single fixed strategy for performance optimization. Optimization of system performance, such as scalability and timeliness, is achieved by means of distribution strategies that are customized for a specific application domain or technical infrastructure. For instance, [51] proposes a specific distribution strategy to obtain scalability of a distributed shared data space across a large number of components in a wide-area network. As a result, those systems are not very flexible. Their use for different classes of applications requires intricate modification of the application code.

Our point of departure is that the trade-off between different extra-functional quality properties can be addressed by a flexible architecture. The flexibility of this architecture consists of the possibility of adapting the distribution policies to application-level characteristics of access to the shared data space. Each distribution policy has its own characteristic trade-off between resource usage and performance optimization. For instance, a distribution policy that replicates data can decrease latency for accessing the data but at the price of a larger amount of used memory (to store the data) and bandwidth (to keep the data replicas consistent). In this way, it becomes possible to provide efficient implementations for a large class of applications. Moreover, the distribution and replication of data items is such that an efficient distributed shared data space can be realized. To this end, we have built a system that realizes distributed shared data spaces in which each data type is distributed and replicated according to a dedicated strategy.

Another important innovation introduced in our system is that the tailorability of
distribution needs of an application is carried out transparently to the application itself. This is in line with the principle of Separation of Concerns (SoC). In this way, it is possible to reuse the same application code in several environments where different distribution strategies are required.

However, identifying before application deployment which distribution policy best suits the application behavior is often very difficult if at all possible. To complicate matters, it might be the case that the behavior of an application changes during its executing time in unanticipated ways.

We make the following contributions. First, we demonstrate how this differentiation of strategies outperforms fixed strategies. We note that differentiating strategies by itself is not new and that it has been applied to distributed shared memory systems [4, 15], and to some extent proposed also for shared data space systems [78]. Second, we show that continuous adaptation of strategies may be needed, which in turn requires a monitoring and feedback system to adjust previously chosen strategies. In this respect, we propose a design that enables the middleware to monitor and subsequently adapt its distribution policy to the actual application behavior. As proof of concept we built a prototype that employs our design. Finally, using the prototype we conduct a series of experiments that prove the benefits of continuous dynamic adaptation of distribution policies.

The chapter is organized as follows. Section 3.2 describes some common distribution strategies and the method that we employ for measuring their performance. Section 3.3 explains our implementation of a distributed shared data space. Subsequently, in Section 3.4 we discuss the results of the experiments that we conducted for validating our approach. In Section 3.5 we discuss several approaches related to our work. We conclude with some final remarks.

3.2 Data Space Strategies

In the past, researchers have sought a solution to the efficiency problem by providing an implementation that was tailored to a specific application domain. Proposed implementations differ in the distribution of tuples. In this section, we describe several strategies that are used for the distribution of tuples in existing implementations of a shared data space. Examples include:

**Statically centralized:** Only a single, fixed node has a local data space where all tuples are stored. This is the common implementation of a non-distributed, but remote accessible shared data space. Examples of systems following this approach include JavaSpaces [28] and TSpaces [94]. This approach has the drawbacks common to all centralized designs. The single node where the data space resides may become a bottleneck under a high load of requests; and it represents a single point of failure.
Ad-hoc centralized: A system that follows this approach is Lime [59]. In Lime each process stores tuples in its local data space. The data space is permanently bound to the process. Processes join and leave the computational environment, together with their local data space. The local data spaces collaborate with each other to give to processes a logical view of a single shared data space. This means that the content of the space dynamically changes when local data spaces join or leave the system. In Lime both processes and data spaces have unique identifiers. If a process has to leave and wants that its tuples are still available, it has to declare the identifier of the destination data space where the tuples should be moved. This peculiarity contrasts the basic principle of the data space model of space decoupling. In fact, to transfer the tuples, the sender has to know the receiver.

Statically distributed: In this case, each tuple is stored at a single node, but different tuples may be stored at different nodes. The node \( n \) responsible for storing a tuple \( t \) is determined by a hash function \( H: n = H(t) \). This approach, with some modifications, has been adopted in the run-time environment developed at York [24]. An advantage of using a hashing function for routing tuple requests is the reduction of searching time. On the other hand, it cannot adapt to changes in the environment.

Fully replicated: Each tuple is replicated to every node. This strategy has been applied in the system described by Corradi, et al. [20]. Since tuples can be found locally, the searching time is negligible. However, this strategy needs sophisticated mechanisms to control the consistency of the data space in the presence of removal operations. Generally, these mechanisms perform poorly when the number of nodes increases.

Structurally replicated: Tuples are replicated according to a structural schema. Carriero and Gelernter propose in [14] a schema that is based on a grid of nodes formed by logical intersecting busses. Each node belongs to one outbus and one inbus. A tuple is replicated in all nodes that form the outbus. Retrieve operations are executed on the nodes of the inbus. Since an inbus intersects all outbusses, the search is extended to the entire data space. Another schema, used in a system proposed by Corradi, et al. [19] follows a tree structure. The leaves of the tree represent the nodes where the processes reside. In the internal nodes of the tree the data space is distributed. A tuple is replicated along the path that starts from the leaf node where the tuple was generated, and goes up to the root node. A search follows a path that starts from a leaf node and may go up till the root node. Also in this case a search is extended to the entire set of tuples stored in the data space, because eventually a searching path will get to the root node where all tuples are stored. Both these approaches scale down the problem of consistency to a part of the data space. However, as the number of nodes increases they incur the same problem of the previous strategy.
3.2 Data Space Strategies

We are interested in an implementation of the shared data space that provides an efficient/acceptable performance under changing application behavior. We make two observations. Firstly, a shared data space system that offers just one global distribution policy is not flexible enough, as it can satisfy the distribution requirements of only a single class of applications. Secondly, in case that several distribution policies are available, it is important to determine which policy matches an application’s needs. It is unclear whether application developers are always capable of making such a decision.

The requirements that we pose for the architecture are the following:

- location transparency: in interacting with the shared data space, the application should not be aware of where tuples are inserted or retrieved.
- separation of concerns: the definition of specific performance concerns should not be part of the application logic.

Crucial in our research is the role that distribution policies play. Distribution policies form our tunable parameter that influences the trade-off between performance and resource usage.

Another issue that needs to be addressed is the data granularity at which a policy is applied. In virtually all shared data space systems, a single distribution policy is applied to the entire data space. In previous research on Web hosting services, it was observed that associating a distribution policy for each Web document separately allows to obtain close-to-optimal performance [61]. In other words, differentiating distribution policies at a finer granularity than an entire data space may be beneficial.

3.2.1 Strategy Evaluation

We hypothesize that ideally, a shared data space system should be supported by multiple distribution policies. Each strategy has specific performance and resource usage characteristics. The shared data space system should automatically determine which policy should be applied to a given application to provide acceptable performance. Moreover, policies should be differentiated at a much finer granularity than the entire data space. If a policy change is needed, a shared data space system should be able to detect such a need and automatically adapt to a better policy.

Next, we need to determine how to quantify the performance of a given distribution policy. The performance of a policy can be expressed in terms of the costs that the system incurs for executing data space operations. For example, full replication may make read operations cheap in terms of latency, but update operations such as put and take may be expensive. Likewise, a higher price needs to be paid for
Figure 3.1 The deployment of GSpace kernels in two nodes. Each kernel consists of two subsystems: the Operation Processing Subsystem and the Adaptation Subsystem.

Some policies do well for some performance metrics, but not well for others. We need a method for comparing policies that allows automatic selection of the best policy. Following the approach described in [62], we adopt the use of a cost function \( CF \) which is a linear combination of \( n \) metrics \( m_{1,p}, \ldots, m_{n,p} \) produced by a policy \( p \):

\[
CF(p) = w_1 * m_{1,p} + w_2 * m_{2,p} + \ldots + w_n * m_{n,p}
\]  

(3.1)

Here, \( w_1, \ldots, w_n \) are weights that determine the relative influence of each metric, that is, \( \sum w_i = 1 \) with \( 0 \leq w_i \leq 1 \). Given a set of weights, the policy \( p \) that minimizes \( CF(p) \) is considered the best. We return to the use of this cost function below.

### 3.3 Architectural Design for Performance

In this section, we propose an architecture, GSpace, that meets the aforementioned requirements. Before diving in the details of our architecture we should clarify what a tuple (or template) is in our system. In our implementation, we used an approach similar to JavaSpaces [28]. Tuples and templates are Java classes. Tuples
and templates used by application components must be subclasses of the Java class `Tuple`.

For each of the modules that are part of the internal architecture of the GSpace architecture we provide a detailed description.

Figure 3.1 shows an example of a component-based application distributed across interconnected nodes that uses GSpace. On each node, a GSpace kernel is instantiated.

A GSpace kernel consists of two subsystems: the Operation Processing Subsystem (OPS) and the Adaptation Subsystem (AS). In the following sections a description of both subsystem is given.

3.3.1 Operation Processing Subsystem

The OPS provides the core functionality necessary for a node to participate in a distributed GSpace: handling application component operations; providing mechanisms for communication with kernels on other nodes; and monitoring connectivity of other GSpace kernels that join and leave the system; and maintaining the information about other kernels. Finally, the OPS provides the infrastructure to differentiate distribution strategies per tuple type. The internal structure of the OPS is shown in Figure 3.2. Internally, the OPS is organized as follows.

The System Boot module is responsible for initiating all the other modules of the GSpace kernel. Subsequently, it advertises its presence to other GSpace nodes in order to establish communication channels and to join a GSpace group.

The Controller provides the following operations.

- `put(tuple)` inserts tuple in the space.
- `read(template)` if there is a tuple in the data space that matches the given template, then a copy of this tuple is returned to the caller. The `read` operation blocks the caller until a tuple matching the query is available in the system.
- `take(template)` is similarly to a `read` operation with the difference that the matching tuple is removed from the space.

The Dynamic Invocation Handler (DIH) determines which distribution policy to apply based on the type and content of the tuple (or template). It operates as follows. If a component invokes a GSpace operation, then the DIH looks up which policy to use for this operation. This information is obtained from a Distribution Policy Descriptor. A distribution policy descriptor is a file containing `(template, distribution policy)`-pairs. Tuples are matched
Figure 3.2 Internal structure of the Operation Processing Subsystem.

against templates to determine which distribution policy to apply. However, to avoid reading a file each time an operation is executed, the information stored in the descriptor is downloaded into the **Policy Table**.

Once the applicable policy has been determined, the DIH ensures that a distribution manager that implements this policy is instantiated. This means that it is possible to instantiate a class and invoke its methods at runtime without knowing the class at the time the code was written. The benefits of this methodology are twofold. First of all, it is not necessary to load at boot time all the available distribution managers, but only the ones requested by a distribution policy descriptor. Moreover, new implementations of distribution managers can be added and used at runtime, without shutting down or recompiling the source code of the GSpace kernel.

The **Distribution Managers** are responsible for enforcing distribution policies. For each distribution policy that the system supports there is a separate distribution manager. Depending on the policy that the manager implements, it may dictate that tuples be sent to or requested from GSpace kernels on other nodes. Communication between distribution managers is realized by

---

1The DIH uses the Java Dynamic Proxy Class mechanism to dynamically instantiate distribution managers and invoke their methods. A dynamic proxy class is a special class created at runtime by the Java virtual machine.
means of dedicated communication module in the kernel. A distribution manager does not make any assumption of of the network topology where is deployed.

The **Data Space Slice** provides a local storage for tuples together with the associative method for retrieving them.

The **Communication Module** provides facilities for sending and retrieving tuples to and from other GSpace kernels. This module provides support for different forms of communication (such as multicasting or point-to-point communication), but also different qualities of service (such as reliable or unreliable communication) to be used for defining different policies.

The **Connection Manager** is responsible for keeping track of information about network locations of other GSpace kernels. This information is stored in the **Address Table** that can be used by distribution managers. To automatically discover other kernels, the Connection Manager uses a discovery mechanism. Currently such discovery mechanism is implemented exploiting the broadcasting facility of a local-area network. However, the GSpace concept is independent of the specific discovery mechanism that is used.

At boot time, the **Policy Descriptor Loader** downloads distribution policy descriptor file. The file is made available to all nodes where GSpace kernels are instantiated. The descriptor may be changed at runtime. Thus, this module monitors any updates of the local descriptor file and reloads it if necessary.

Next, we provide a description of the Adaptation Subsystem.

### 3.3.2 Adaptation Subsystem

The AS is an optional addition to GSpace that provides the functionality needed for dynamic adaptation of policies. The AS communicates with the co-deployed OPS for obtaining information about the status and actual usage of the system. Periodically, the AS analyzes this information and evaluates the system performance. Based on this information, the AS may decide to change to another distribution policy.

Figure 3.3 shows the internal structure of an AS. It consists of the following modules:

- **Logger** is responsible for logging all the space operations executed on the local kernel. When the OPS receives a request for a space operation from an application component, it informs the Logger about the operation. The Logger keeps track of the number of operations that have been executed...
for each tuple type. When the number of operations for a particular type reaches a threshold, the logger notifies its local *Adaptation Module*. Each tuple type in the system might be associated with a specific threshold.

The *Adaptation Module (AM)* is the core of the Adaptation Subsystem. The AM is responsible for deciding when the different phases of the *adaptation mechanism* should be started. The code of the AMs on all nodes is identical. However, for each tuple type in the system one AM operates as a *master* and all other AMs operate as *slaves*. The master AM is responsible for the adaptation decisions for a particular tuple type. The slave AMs follow the decisions taken by the master. Because the AMs on all nodes are identical, it is in principle possible for any slave to take over the role of master if the latter leaves the system.

The selection of the kernel in which the AM works in master mode is made through a simple election algorithm. When a GSpace kernel is instantiated, the kernel stores its *instantiation time* (that is, the actual time provided by the host node). During the discovery protocol each kernel sends its instantiation time. Each kernel stores in the Address Table the information about the other kernels that have been discovered. This information includes the address of the kernel, its status and the instantiation time. The entries present in the Address Table (included the one for the local kernel) are ordered according to the instantiation time of each kernel. This order is the same in the Address Tables of all kernels. The *oldest* kernel in the system (that is the kernel with the earliest instantiation time) is elected as the kernel.
where the master AM is located. Note that it is not necessary to provide any
clock synchronization across the nodes. This algorithm makes the assump-
tion that the kernel that has been running for the longest time is deployed
on the node with the highest availability in the network. However, in case
that two or more kernels show the same instantiation time then the address
value can be added to the instantiation time to provide a unique value (since
the address value is unique for each node). If the node where the master
AM is allocated leaves the system, all the other kernels are informed and
the address of that node is withdrawn from each Address Table. Then the
address of the new master AM is the address that occupies the first position
in the reordered Address Table. Since all the AMs are the same, the selected
AM switches from slave to master mode immediately and can fulfill all tasks
as the previous AM. The tuple types that applications are going to use are
listed in the policy descriptor file. A copy of this file is made available on
each kernel, in which tuple types are listed in the same order in all nodes.
Hence, for the \(i\)-th tuple type in the file the master is the node that appears
in the \(i\)-th position in the address table. However, because the number of
tuple types can be greater then the number of AMs in the system, an AM
is able to be assigned as master of more than one tuple type.

The Cost Computation Module (CCM) performs runtime simulations using
the logs. It obtains the logs from the AM. For all operations in the log it
asks the DPCM (described next) to provide the cost of execution of this
operation. The CCM aggregates the cost over a complete log. The CCM
passes the results of this simulation to the AM.

The Distribution Policy Cost Models (DPCM) has the task to compute
the cost incurred by the corresponding distribution policy for a given log of
operations. In order to enable adaptation, a distribution cost policy model
must be provided for every distribution policy available to the GSpace sys-
tem. When a runtime extension of the suite of distribution policies available
to a GSpace system is required, a DPCM must be provided for every new
distribution policy. The new DPCM(s) can also be loaded at runtime in the
system.

When the distribution policy for a tuple type is adapted, it is possible that tuples
of that type are present in the shared data space. We refer to these tuples as
legacy tuples. A Transition Policy prescribes how to handle legacy tuples
in order for them to be placed at locations where the new distribution policy
expects to find them. For each tuple type, the application developer can
specify which transition policy to apply.

Adapt-Comm Module (ACM) provides communication channels between the
ASes on different nodes in the system. Creating a dedicated module for the
communication of the ASes is just a design decision. This leads to a cleaner
design then having a single module handling all communications.
In the following section, we take a look at the mechanism that allows GSpace to select the best strategy that suits the application behavior.

### 3.3.3 Autonomic Behavior of GSpace

This section is dedicated to the adaptation mechanism that allows GSpace to select the best distribution policy for a given tuple type during runtime. These actions can be grouped into three phases.

#### Logging Phase

The first phase is called *logging phase*. During this phase, statistical data is collected about the operations that application components perform for each tuple type. Based on the data collected during this phase, the system will determine the distribution policy that best fits the application distribution requirements for a given tuple type. In Figure 3.4 a message sequence chart shows the actions executed during this phase. The Controller, who receives the requests for space operations from the application components, passes the data about the current operation to the Logger. This data contains:

- Operation type: the space operation executed (either a `read`, `take` or `put`)
- Tuple type: the type of the tuple or template passed as argument with the operation
- Location: the address of the kernel where the operation is executed
- Tuple ID: a unique id associated with each tuple that enters the shared data space

---

**Figure 3.4** The MSC of the logging phase.
3.3 Architectural Design for Performance

3.3 Architectural Design for Performance

Send the local logs for the tuple type

Calculate the costs for each policy using the respective model

Pass the costs to the Cost Computation Module

Pass the predicted costs for each policy to the Adaptation Module

Generate the predicted costs for each metric

Compare the CF that each policy produces and select the one that minimizes the CF

Check if the actual policy is the best policy

Figure 3.5 The MSC of the evaluation phase.

- Tuple size: the size of the tuple inserted through a put operation or returned by a read or take operation
- Template size: the size of the template passed as argument of a read or a take operation.
- Timestamp: the time when the operation is executed

Once the number of operations for a tuple type reaches a threshold in one of the nodes, the system switches to the evaluation phase.

Evaluation Phase

The exchange of messages during the evaluation phase is shown in the message sequence chart in Figure 3.5.

In this phase the master AM asks all slave AMs to report their local logs for the tuple type. The timestamps in the operation logs are compensated for clock drift. Subsequently, when all logs are gathered by the master, the CCM at that node sorts the aggregated log in chronological order.

For each distribution policy available in the kernel at the time when the evaluation phase is executed, the CCM feeds the logs to the respective DPCM. The DPCM computes the costs that the system would have incurred if that distribution policy
had been applied to the tuple type. The CCM collects the costs from the DPCM and passes these to the AM. The AM combines the predicted costs for each policy in a cost function value (more on this in Section 3.3.4). The AM compares those values and selects the best policy, that is the one that minimizes the cost. The AM checks whether the policy currently associated with the tuple type (it retrieves this information from the policy table) is also the best policy. If this is the case, no further actions are undertaken. Otherwise, the AM starts the Adaptation phase.

If the node in which the threshold was reached is not the node where the master AM is allocated, then the slave AM notifies its master. After that, the master AM proceeds to the evaluation phase as explained above.

Adaptation Phase

Figure 3.6 shows the message sequence chart for the adaptation phase. The master AM starts by freezing the operations for the tuple type in the system. This means that during the adaptation phase, the adaptation subsystem will block all incoming requests from application components.² The master AM updates its local policy

²Notice that this solution of freezing system’s activity was taken just for its simplicity and easiness to implement.
table and then commands all slaves to update their local policy table. The update consists of setting the best distribution policy as the distribution policy for the tuple type.

At this point, each AM retrieves the transition policy associated with the tuple type. The transition policy takes care of possible legacy tuples present in the local data space. Subsequently, the master unblocks the operations for the tuple type and normal activity is resumed.

### 3.3.4 Distribution Policy Evaluation

Next we discuss the method used in the evaluation phase to quantify the performance of each distribution policy.

Currently, a number of distribution policies are available for GSpace. Each policy strikes a different balance between tuple access time and resource usage. Together with the patterns of tuple accesses by application components these factors determine the performance of a distribution policy.

To compare the performances of distribution policies we follow an approach inspired by [62]. We define a *cost function* as a linear combination of metrics that capture different aspects of the costs incurred by a policy. The cost function combines these costs in an abstract value that quantifies the performance of a distribution policy. We used the following metrics in the cost function: \( rl \) and \( tl \) represent the cumulative latency for the execution of read and take operations, respectively; \( bu \) represents the total network bandwidth usage; and \( mu \) represents the memory consumption for storing the tuples in each local data space. The performance metrics are such that a lower value indicates a better performance. For these parameters, the cost function for a policy \( p \) becomes:

\[
CF(p) = w_1 \times rl_p + w_2 \times tl_p + w_3 \times bu_p + w_4 \times mu_p
\]  

(3.2)

Because put operations are nonblocking, application components do not perceive any difference in latency for different distribution policies. Therefore, the put latency is not used as a parameter for the cost function. The burden incurred because of put operations is captured by the latency experienced by read and take operations, which are included in the cost function. The \( w_i \)’s control the relative contribution of individual cost parameters to the overall cost. For instance, if the latency of the operations is crucial for the performance of the application, the application designer can assign higher values to the weights associated with those metrics.

The conclusions of our experiments do not depend on a specific setting of these factors. For the experiments in this paper, we take \( w_i = 0.25 \) for all \( i \).

Periodically, the master AM for a tuple type evaluates the cost for each distribu-
tion policy. These evaluations are performed by means of simulation using policy models.

Currently in GSpace the following distribution policies are available: Store locally (SL), Full replication (FR), Cache with invalidation (CI), and Cache with verification (CV):

**Store locally (SL):** A tuple is always only stored on the slice that executes its put operation. Likewise, read or take operations are performed locally as well. If the tuple is not found locally then a request is forwarded to other nodes.

**Full replication (FR):** Tuples are inserted at all nodes. The read and take operations are performed locally. However, a take has to be forwarded to all nodes by means of a totally ordered broadcast, in order to remove all copies.

**Cache with invalidation (CI):** A tuple is stored locally. When a remote location performs a read operation, a copy of the tuple is subsequently cached at the requester’s location. When a cached tuple is removed through a take operation then an invalidation message is sent to invalidate all other cached copies of that tuple.

**Cache with verification (CV):** This policy is similar to CI, except that invalidations are not sent when performing a take. On reading a cached tuple, the reader verifies whether the cached copy is still valid, that is, whether the original has not been removed.

For each of these policies, we developed the respective DPCM. Thus, in each kernel we have the following policy models: SL-DPCM, FR-DPCM, CI-DPCM, and CV-DPCM.

The DCPM contains a model of a specific policy. This model predicts the cost for executing a data space operation. This cost is expressed in terms of the variables that occur in the cost function (latency, bandwidth use and memory use). For each DPCM, the CCM iterates through the logs and for each log the CCM uses the respective operation model.

As an example, Figure 3.7 shows the pseudo-java code for the readModel operation in the SL-DPCM. The operation takes two parameters: 1) the set of logs for put operations that insert tuples that the read operations can match (this set of logs is maintained by the CCM), and 2) the log for the read operations.

According to the Store-locally policy, the read operation first has to search on the local node for a matching tuple. In lines 4-8, the readModel iterates through the set of logs of put operations searching for a put executed on the same location as the read operation. If such a put has been logged, then the read can return a copy of the matching tuple. In this case, just the latency for accessing the local data space is accounted (line 8). Otherwise, the read operation has to send the request to the
readModel(Vector putLogs, Log readLog) {
  //iterate through the vector of logs for put operations
  while (putLogs.hasMoreElements()) {
    put = putLogs.next();
    //if the put was executed on the same node of the read
    if (put.location.equals(readLog.location)) {
      readLatency += Profiler.readLocalLatency(readLog.templateSize);
      return;
    }
  }
  //iterate through the vector of all known nodes
  while (addrTable.hasMoreElements()) {
    msgSize = Profiler.sendingPacketSize(readLog.templateSize);
    bandwidthUsage += msgSize;
    readLatency += Profiler.networkLatencyTCP(msgSize);
    addr = addrTable.next();
    //iterate through the vector of logs for put operations
    while (putLogs.hasMoreElements()) {
      put = putLogs.next();
      if (addr.equals(put.location)) {
        msgSize = Profiler.sendingPacketSize(put.tupleSize);
        bandwidthUsage += msgSize;
        readLatency += Profiler.networkLatencyTCP(msgSize);
        return;
      }
      msgSize = Profiler.nullReplyPacketSize();
      bandwidthUsage += msgSize;
      readLatency += Profiler.networkLatencyTCP(msgSize);
    }
  }
}

Figure 3.7 The readModel operation in the SL-DPCM.
other kernels. As for the real operation, the readModel goes through the addresses in the address table in search of the location of a matching tuple (lines 13-21). For each request sent to a node, the readModel accounts the bandwidth usage (line 15). This value is given by the size of the requested message. The message contains the header and the payload, which contains the size of the template, given as argument to the read operation (line 14). Furthermore, the network latency for sending this request using TCP is accounted (line 16). If a put operation has been logged in the current location (meaning that a matching tuple is in this node), then a copy of the tuple is returned. The readModel accounts the bandwidth for sending the reply message with the matching tuple (line 23-24) and the latency for sending the message back to the requester (line 25). If in the current location no put operations have been logged, then a message with a null reply is sent back to the requester. Also in this case, the readModel accounts the bandwidth usage (line 28-29) and the network latency (line 29).

3.3.5 Automatic Profiling

Information about the latency for network accesses and for local data space accesses is provided by the profiler module. When GSpace is deployed for the first time in a new environment, the profiler creates a file that contains the network and data space profiles.

For the network profile, the profiler sends a number of packets of different sizes to a remote echo server\(^3\) (for both TCP and UDP packets) and measures the time for the round-trip. This data is used for building a function that for a given packet size returns the latency for sending the packet.

For profiling the access to the local space, the profiler executes a number of read and take operations on a local data space with templates of different sizes measuring the time to complete each operation. The data collected is used for building a function that for a given template size returns the access latency.

The parameters to build those functions are stored in a file, called profile.inf. At boot time, the system tries to load the file. If the file is present, then the environment was already profiled. Otherwise, the profiler of a kernel is chosen to start the profiling phase. Once the necessary data has been collected and processed, the profiler stores the data in the profile.inf file and makes the file available to the other kernels in the LAN. In this way, the modules are automatically calibrated each time the system is deployed in new environment. In case the hardware where the system is deployed changes, such as when the nodes are interconnected via a faster network, a re-calibration must be forced.

\(^3\)More precise model can be build by pinging more than one remote server and averaging the results.
3.3.6 Adapting the Data Space Content

This section describes the actions that are taken when the data space content needs to be adapted due to a change of distribution policy.

According to the semantics of read and take operations, when a matching tuple is inside the shared data space it should be returned. Since GSpace is a distributed shared data space, each distribution policy has its own strategy for searching for a matching tuple across the nodes during a read or take. This strategy is influenced by the modality in which tuples are inserted through the put operations of that distribution policy. When the system changes the policy associated with a tuple type as consequence of an adaptation, it is most likely that legacy tuples are still inside the data space. Potentially, legacy tuples could be located at a different location than where the new policy will look for. If the searching strategies of the old and the new policy are different, then without further measures the system cannot guarantee that a matching tuple inside the shared data space is always returned. Depending on the particular application, it could be the case that those tuples could be ignored since new tuples will be soon available. However, this is not the case in general.

For this reason we introduce the transition policy. A transition policy lets the application designer specify the actions to take for the legacy tuples of a given tuple type when an adaptation is performed. If for a given tuple type the transition policy is not specified, a Default Transition Policy (DTP) is available. This DTP removes all legacy tuples and reinserts them according to the new policy. This ensures that the space is kept consistent, and reduces the effort of the developer of distribution policies (who does not have to invent a transition policy). The default policy may be costly. To provide the possibility to reduce this cost, GSpace provides the option to define specific policies for making a transition from existing policies to the new policy.

As an example, let us assume that the system has to change policy from SL to FR. The read and take operations in FR search in the local data space for a matching tuple since all tuples are replicated. If a tuple is not found in the local data space then matching should fail. When switching from SL to FR, we need to guarantee that these operations will behave correctly. Therefore, upon switching policy, it is necessary to replicate the legacy tuples to all local data spaces. The DTP simply first removes all legacy tuples and subsequently reinserts them according to, in this case, the FR policy. As a result, tuples are replicated across the entire system.

The execution of a transition policy may involve extra costs. These costs should be taken into account when switching policy. Depending on the number of legacy tuples that needs to be reinserted, the costs of redistribution could be too high compared to the actual gain that the system achieves by adopting the best policy. However, for a long period of execution the best policy may reduce the overall costs to such a level that the extra costs for the redistribution actually pay off. This
problem falls in the category of Online Decision Making with Partial Information problems, of which the *The Ski Rental Problem* is a classic formalization [35].

### 3.4 Implementation and Experiments

In this section we present the results of the experiments that we conducted using our implementation of GSpace. These experiments provide answers to the following questions. First of all, does it make sense to have several distribution policy? Or in another words, is there a policy that fits all the application needs? The second question regards those dynamic applications that show a changeable behavior during execution: can such an application gain in performance by dynamically adapting the distribution policy to the actual application needs? Finally, we address the question whether the gain that the adaptation produces is enough to compensate the overhead that is introduced.

Using the same set-up, we run several sets of experiments that aim to answer these questions. The first set aims to show that there is no single distribution policy that can perform optimally for different application behaviors. This is especially true when the application behavior changes during execution. As a consequence, a significant gain in performance can be achieved through the use of dynamic adaptation of data distribution policies. This is shown in the second set of experiments. The final set of experiments provides measurement of the overhead introduced by the adaptation mechanism and it shows that this overhead is small compared to the benefit that the system gains.
3.4 Implementation and Experiments

3.4.1 Application Model Description

In this section, we discuss the application model that is used for the experiments. The experiments consist of executing an application composed of several components that exchange data via GSpace. The application is designed to simulate several application usage patterns. A usage pattern of an application is characterized by the order and the ratio in which the application’s components execute data space operations. In the following, we list the application usage patterns used in the experiments.

Local Usage Pattern (LUP): In this case, tuples are retrieved from the slice on the same node where they have been inserted. This could be the case if components store some information for their own use or if producer and consumer of a tuple type are deployed on the same node.

Write-many Usage Pattern (WUP): In this usage pattern applications on different nodes need to frequently and concurrently update the same tuple instance. This is problematic for the consistency of distributed shared-memory systems, since mutual exclusion is required.

Read-mostly Usage Pattern (RUP): In this usage pattern, application components execute mostly read operations on remote tuples. We distinguish two variants of this pattern: 1) RUP(i), where applications may execute tuple updates between sequences of read operations. An example could be of a tuple type representing a list-of-content. 2) RUP(ii), between the insertion of a tuple and its removal only read operations are executed. This could be the case of a tuple type representing intermediate-result data in a process-farm parallel application.

Figure 3.8 shows the experiment setup. On each of several nodes one GSpace kernel is instantiated together with one application component. All application components are equal, except for the component called Coordinator. The coordinator is a special component that acts as a conductor in an orchestra, directing and coordinating the actions of each application component. The coordinator resides in one of the nodes used for the experiment. It is connected directly with each component, thus communication between coordinator and application components is external to GSpace.

During an experiment run, the coordinator executes the following steps:

1. generate a sequence of data space operations with a certain ratio of put, read and take operations
2. generate the schedule in which components have to execute the operation sequence
3. for each policy do:

(a) execute the schedule, dispatching each operation in the sequence to the components

(b) change the distribution policy associated with tuples, reset the data space content.

During the experiments, we measured the actual values of costs that each distribution policy produced. The same sequence and the same schedule are repeated for all distribution policies. This ensures that the comparison between the distribution policy is unbiased by randomization effects.

3.4.2 Implementation Measurements

This section focuses on the first set of experiments. In this set, we tested each application usage pattern with the currently available distribution policies in GSpace. Sensors were placed in the system to measure the following cost parameters at run-time:

- The latency for the execution of a read operation ($r_l$)
- The latency for the execution of a take operation ($t_l$)
- The network bandwidth usage ($b_u$)
- The total amount of memory consumption for storing the tuples in each local data space ($m_u$)

The results that are shown in this section were obtained by actual measurements. For this specific set of parameters, the cost function (for policy $p$) becomes:

$$CF(p) = w_1 * r_l + w_2 * t_l + w_3 * b_u + w_4 * m_u$$  

Recall that, since put operations are executed without blocking the application components, we do not use the latency of put operations as a parameter for the CF. Also, for the calculation of the CF values we use the same value for all weights $w_i$, that is 0.25.

The optimal policy for an application usage pattern is the policy that produces the lowest CF value. It should be noticed that the proposed approach implies that the optimal policy is identified through a comparison of their execution performance, rather than via a prediction.

All experiments were executed on 10 nodes of the DAS-2 cluster [5] allocated exclusively for a GSpace kernel and an Application Component. Each DAS-2
For each application usage pattern, a histogram shows the cost incurred by different distribution policies. Note the log scale for RUP(i) and RUP(ii).

Figure 3.9-(a) shows the results collected when the LUP was simulated. Under these conditions, the policy SL produces the lowest CF value. In fact, SL guarantees low cost for the execution of space operations that take place on local tuples. Instead, other policies provide more sophisticated strategies to distributed tuples at the cost of using more resources.
Figure 3.10  Results of the simulation for the RUP(i) with different run lengths.

Figure 3.9-(b) shows the results obtained when the WUP was simulated. In this scenario, several components need to access and modify the same instance of a tuple. The lowest CF value was output by policy FR. In fact, the extra resources needed for replicating tuples lower the access time, since no search time is required for finding a matching tuple. The other policies perform worse, since for all of them read and take operations require a global search when the tuple is not stored locally.

The results in Figure 3.9-(c) and (d) are obtained when the RUP(i) and RUP(ii) were simulated. The values in the histograms are reported in logarithmic scale. In both scenarios, application components execute multiple reads on the same tuple instance and a few take operations. In both cases, the lowest CF values were produced by the CI strategy, since caching allows to execute most of the read operations locally. However, policy CV performs always considerably worse than policy CI due to the validation message that the reader has to send for each operation executed on the cache. Thus, both latency time and bandwidth usage for read operations resulting higher value than for CI. The performance of the FR policy is close to that of CI. However, the FR strategy uses more memory for storing replicas in all node; whether CI stores the cached tuples only in those nodes where they were requested.

The results of the experiments show that no single policy can perform optimally for all different application behaviors. This means that it is crucial to find out which policy fits the application behavior to optimize the performance of the system.
3.4.3 A Case for Dynamic Adaptation

Figure 3.10 shows some unanticipated results collected for a set of experiments with the RUP(i). Here, the ratio of number of read operations to number of take operations has a lower value (meaning that more take operations are executed) than in the experiment in Figure 3.9-(c). The X-axis shows the length of the run, that is, number of operations. The Y-axis shows, on a logarithmic scale, the cost incurred by the distribution policies. The experiments described before suggest that the best policy for RUP(i) is CI. Instead, the graph shows that only for shorter runs, cost is minimized by the CI policy. As the number of the operations increases policy FR outperforms policy CI.

The reason for this behavior is due to the increased number of take operations executed for each run. This fact has two effects that jeopardize the performance of policy CI. Firstly, the execution of more take operations reduces the benefits introduced with caching since cached tuples are more often invalidated. Thus, read operations have to search for a matching tuple, increasing latency time and bandwidth use. On the other hand, policy FR replicates tuples at every insertion thus replicas are already available locally. Secondly, for each take operation policy CI uses point-to-point messages for cache invalidation. Instead, policy FR exploits the more effective atomic multicast technique for removing replicas, that reduces resource usage.

What we see is that even given the behavior of an application, it is difficult to predict which policy it fits best. One solution is to make more accurate models for predicting the cost of policies from behavior. Building these models is quite intricate. For one thing, it is quite complex to determine all the parameters needed for such a model. An alternative approach is to let the system itself figure out which policy works best.

The experiments presented here show that there is a benefit in having different distribution policies. This characteristic makes the system flexible enough to cater efficiently for different application usage patterns. Thus, it is possible to minimize the resource usage for tuple distribution when the distributed policy that fits the needs of the application is used. Secondly, the unpredictable behavior discussed here stresses the importance of the impact that GSpace’s adaptability has at application level. In designing GSpace we adopted the SoC principles to keep extra-functional concerns (such as data distribution) separated from application functionality. This allows us to change a distribution strategy without the need to modify one single line of application code. As a consequence, the system can switch from one distribution strategy to another at runtime, without interrupting the application execution.

In [62] an approach is reported in which a system automatically selects the best strategy for caching Web pages. This approach works by internally replaying and simulating the recent behavior of the systems for a set of available strategies.
Based on these simulations, the system can decide which policy works best for the current behavior of the system. This approach was also implemented in GSpace and the results of the experiments discussed in the following prove the benefit of implementing such an approach.

### 3.4.4 Performance and Overhead

In this section, we discuss the results of the second set of experiments. These experiments focus on two aspects of the adaptation subsystem: the performance gain and the overhead that is introduced when the adaptation is activated.

To measure the performance gain when adaptation is used, we executed the following experiments. We produced a set of runs in which the behavior of the application model changes. In each run, at least 500 operations are executed according to the same application usage pattern. We refer to this part of a run where the same application usage pattern is used as a run phase.

Firstly, we instantiated GSpace without the adaptation mechanism. For each policy we executed all runs, collecting the operation logs. At the end of the run, we executed the simulation on the logs for each policy, obtaining the cost function values for each policy. Out of these values, we selected the best cost function values. Subsequently, we executed the same runs but this time GSpace used the
adaptation mechanism. We employed different threshold values used for triggering the evaluation phase, expressed as the number of operations before a next evaluation takes place. Every time the evaluation phase was terminated we stored the best cost function value. In the end of the execution these values were summed together, producing an aggregated cost function value. This value represents the total cost incurred during the execution of the run with the adaptation.

Figure 3.11 shows the graph where the threshold values are placed on the $X$-axis and on the $Y$-axis the cost function values. In the graph, the aggregated cost function values for different thresholds are compared with the best cost function value produced during the first phase of the experiments. For all threshold values, the performance of the system with adaptation outperforms the performance of the policy that performs best without adaptation. In particular there is a gain of 30% when the threshold is 50, which reduces to 5% when the threshold is 2000. The graph shows that the smaller the threshold the better the performance. When the threshold value is much smaller than the length of a run phase, the system can detect more quickly when there is a change in application usage. Hence GSpace can decide sooner to switch to the best policy. Therefore, the total aggregated costs for small threshold values are lower than the costs for large threshold values.

For threshold values larger than 2000 the performance of the system with the adaptation mechanism activated is similar to that of the system without adaptation. This is explained by the fact that with large thresholds the adaptation mechanism evaluates over a collection of logs that contains multiple run phases. This means that the mechanism cannot adapt the distribution strategies to the application behavior changes in time leading to a performance similar when the adaptation mechanism is not engaged at all.

This is also confirmed by the graph in Figure 3.12. This graph shows on the $Y$-axis the percentages of executed adaptations (change of policy) for each threshold value ($X$-axis). For small threshold values the system has to adapt less often, since once the best policy is determined it has to change only during the next run phase. As the graph shows, the percentage of adaptations per evaluation increases up to a threshold-value of 1000 and then starts to decrease. This decreasing trend can be explained in the following way. At the increasing of the threshold the number of operation logs that are used in the simulation for the evaluation phase increases as well. In such a large number of operations, more run phases are captured, leading to a random usage pattern behavior. With such a behavior, the policy that performs the best is Full Replication as shown in Section 3.4.2. Since the adaptation mechanism selects this policy most of the time the number of adaptation phases that the system executes starts to reduce. This explanation is also supported by the fact that the performance of the system for larger threshold values is very close to the static case, where the cost function value is obtained by the Full Replication policy (see Figure 3.11).

The costs incurred by doing adaptation comes from two factors:
Figure 3.12 The percentages of adaptation phases compared to the number of evaluation phases for different threshold values.

<table>
<thead>
<tr>
<th>Threshold</th>
<th>50</th>
<th>100</th>
<th>250</th>
<th>500</th>
<th>750</th>
<th>1000</th>
<th>1500</th>
<th>2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total evaluation time</td>
<td>201</td>
<td>192</td>
<td>192</td>
<td>192</td>
<td>193</td>
<td>133</td>
<td>195</td>
<td>193</td>
</tr>
<tr>
<td>Total transition time</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>6</td>
<td>0.6</td>
<td>1</td>
<td>0.02</td>
<td>0.2</td>
</tr>
<tr>
<td>% of total exec. time</td>
<td>15%</td>
<td>14%</td>
<td>14%</td>
<td>14%</td>
<td>14%</td>
<td>9%</td>
<td>14%</td>
<td>14%</td>
</tr>
</tbody>
</table>

Table 3.1 The evaluation time, transition time and their percentage respect to the total execution time. Time is in seconds.

1. the costs in performing evaluation. This leads to additional network traffic for collecting logs and to additional computation time for simulating the policies for the logs.

2. the costs of making a transition from one policy to a new one. These costs depend on the particular transition policy.

For the default transition policy (DTP) we performed a number of measurements. Table 3.1 shows the time needed for evaluation and transition to a next policy for increasing thresholds.

From the values of the first row we infer that the threshold value does not influence the total evaluation time. This can be explained in the following way. The total evaluation time ($TET$) is equal to the time spent for a single evaluation ($SET$) times the number of evaluation executed ($NE$):
3.4 Implementation and Experiments

\[ TET = SET \times NE \]  
\( (3.4) \)

The time spent in a single evaluation is equal to the time for the evaluation of a single operation \((SOT)\) times the number of operation logs collected. The number of operations logs collected is given by the threshold used:

\[ SET = SOT \times Threshold \]  
\( (3.5) \)

The number of evaluations, \(NE\), is equal to the total number of executed operations \((TEO)\) divided by the threshold and is given as:

\[ NE = \frac{TEO}{Threshold} \]  
\( (3.6) \)

Substituting 3.5 and 3.6 in 3.4 we obtain:

\[ TET = SOT \times TEO \]  
\( (3.7) \)

From Formula 3.7 it is clear that the total evaluation time is independent from the actual threshold value.

Notice that during the evaluation phase on the master node, the system is still able to serve the application requests. Only during the transition phase the system does not accept requests until termination of this phase in all nodes. However, the duration of a transition is hard to predict because it depends on the number of tuples that are still in the local slices of the kernels when the adaptation takes place. From the results we can see that small threshold values have a higher chance of finding legacy tuples that need to be moved to a new location. Although a small threshold value has the advantage of increasing system performance, it might increase the cumulative time waiting for transitions to complete.

Finally, the last row in the table shows the percentage of the total time spent in evaluating and making transition with respect to the total execution time of the run. This extra 14\% overhead due to the adaptation mechanism is worthwhile to pay compared to the gain in performance that the system achieves.

3.4.5 Accuracy of the Model

As we explained in Section 3.3, the adaptation mechanism uses models for predicting the metrics used for the calculation of the cost function. If these models are not accurate then the mechanism might take wrong decisions. These decisions could lead to the selection of the wrong \textit{best} policy jeopardizing the performance of the system.
For evaluating the accuracy of our models we performed the following experiments. We executed several runs of operations. For each run, we collected both the values measured using the sensors and the values predicted by the models.

The histograms in Figure 3.13 show a comparison between the actual and predicted cost values for each policy. These values are representative of the complete set of data that we collected during our experiments. The values indicate that the prediction of the model is extremely close to the actual measurements.

In this example, it is worth noticing the result of the selection for the best policy, which occurs during an evaluation phase. Policy SL and policy CV perform quite similar according to both actual and predicted values. Still, the selection of the best policy according to the model is not the same as for the actual measurements. According to the actual measurements, the policy with the lowest cost function value is policy CV. However, the policy with the lowest cost predicted by our model is policy SL. Thus, the adaptation system (that uses predicted costs) would have chosen policy SL instead of policy CV, leading to a sub-optimal solution. However, this sub-optimal choice is not so dramatic since wrong policy performs close to best policy (for this specific case the loss is less than 1%).
3.4.6 Choosing the Appropriate Threshold

The basic assumption behind this research is the possibility to predict the future behavior of an application through the analysis of its near-past behavior. Research described in [62] proves that this assumption holds for Web applications. Although we have achieved promising results in our own work, we need to use real-world traces to validate our basic assumption.

Related to the prediction of application behavior is the selection of an appropriate threshold value for a given application. If the threshold is too small then the system will not have enough information for a correct prediction. In addition, adaptation may introduce a high overhead to the entire system. In contrast, a larger threshold reduces the frequency of evaluation at the cost of lower performance due to late identification of a better distribution policy.

The correct threshold should be such that the system gain introduced through adaptation balances or improves the overhead of the adaptation itself. For example, consider Figure 3.11. For the given run, the choice of the threshold can make a difference between 30% and 5% performance improvement in comparison to a static deployment. However, the adaptation itself introduces 14% overhead. To be effective, the performance improvements should outweigh the overhead costs. In our case, the threshold value should be selected between 50 (with an effective gain of 16%) up to 500 (with an effective gain of 5%).

A general solution to selecting a threshold could be to introduce a warm-up period in which an application deploys a default distribution policy, and logs are collected to capture its behavior in an attempt to identify run phases. Using the techniques described previously, we can then find a reasonable threshold value from which point adaptive distribution is deployed. Note that a similar continuous external analysis of an application’s usage patterns would allow us to even dynamically adjust the threshold value. We consider these matters to be future research.

3.5 Related Work

3.5.1 Shared Data Spaces

Several different approaches for realizing shared data space systems have been proposed. The most common approach is to build a centralized data space in which all tuples are stored at a single node. The main advantage of such an approach is its simplicity. Examples of this approach include JavaSpaces [28] and TSpaces [94]. The obvious drawback is that the single node may become a bottleneck for performance, reliability and scalability.

For local-area systems, a popular solution is the statically distributed data space,
in which tuples are assigned to nodes according to a system-wide hash function [68]. Static distribution is primarily done to balance the load between various servers, and assumes that access to tuples is more or less uniformly distributed across nodes and across time. With the distributed hashing techniques as now being applied in peer-to-peer file sharing systems, hash-based solutions can also be applied to wide-area systems, although it would seem that there is a severe performance penalty due to high access latencies.

The shared data space has been used also in highly dynamic environments, such as in home networks. Those environments are characterized by devices that unpredictably join and leave the network. An approach for coping with such dynamic environments is to dynamically distribute the data space. A system that follows this approach is Lime [59]. In Lime, the shared data space is divided into several transient data spaces that are located on different nodes that form a network. The content of the shared data space changes dynamically upon connection and disconnection of devices. Tuples generated on a device are stored in the local transient data space. When a device connects to the network, the content of its local data space is made available to the entire shared data space. If the device is disconnected the content of its local data space is no longer available unless special actions are taken upon departure time.

A somewhat similar yet simpler approach is followed in SPREAD [21], which is a shared data space system tailored towards mobile and embedded computing. SPREAD follows a store-locally strategy and take operations can be performed only by the node that stored the tuple. However, read operations can be carried out by any node that is in range of a tuple.

Fully replicated data spaces have also been developed, as in [20]. In these cases, which have been generally applied to high-performance computing, each tuple is replicated to every node. Since tuples can be found locally, search time can be short. However, sophisticated mechanisms are needed to efficiently manage the consistency amongst nodes. The overhead of these mechanisms limits the scalability to large-scale networks.

Eilean [78, 13] is a distributed shared data space system that explicitly addresses scalability issues. Together with GSpace, Eilean is the only example of a shared data space system that provides multiple tuple distribution policies. Like GSpace, Eilean is able to differentiate distribution policies on a per-tuple-type basis.

In contrast to GSpace, the tuple-distribution policy association in Eilean can only be statically defined as part of the application. The programmer uses his knowledge of the application access pattern to define the association. In this chapter, we demonstrated that this static association is not enough for providing an efficient distribution of tuples. With the adaptation mechanism described in this chapter, GSpace is able to monitor the application behavior and dynamically adapt the distribution policy for each tuple type. Another difference between Eilean and
3.6 Conclusion

GSpace is that in GSpace the set of distribution policies can be extended and new distribution policies can be downloaded in the system even during execution.

3.5.2 Adaptive Shared-object Systems

We are not aware of any shared data space systems that are able to dynamically adapt to the application needs. Systems with this type of adaptive capability do exist in the domain of shared objects.

One of the first systems that adopted a form of automatic differentiation was Orca [4]. This system provides support for physically distributed objects. An object can be in one of two forms: fully replicated or as single copy. By monitoring the read-write ratios, the run-time system can dynamically switch an object between the two forms.

Further differentiation is offered by fragmented objects [46], and Globe’s distributed shared objects [79]. Both systems separate functionality from distribution aspects by subdividing objects into at least two subobjects. One subobject captures functional behavior and can be replicated across multiple nodes. Each copy of such a subobject is accompanied by a subobject that dictates when and where invocations can take place, similar to the role of distribution manager in GSpace (as part of the OPS [71]). The main difference between GSpace and these two systems, is GSpace’s more evolved approach towards runtime adaptations. With fragmented objects, distribution strategies were more or less static; in Globe, dynamic adaptation has only been partly implemented.

For sake of completeness, we also mention the support for differentiating distribution in distributed shared memory systems, notably Munin and later Treadmarks (for an overview, see [65]). In these cases, distribution strategies have mostly been static and needed to be fixed at compile time.

3.6 Conclusion

The contributions of this chapter are the following.

First, we presented a framework based on the shared data space model, that separates the specification of performance concerns from the application functionality. The framework has a basic suite of distribution policies. Additionally, this suite of policies is extendable. New policies can be built and deployed in the framework.

Second, we demonstrated by means of experiments that:

- this differentiation of strategies outperforms fixed strategies; and
- a significant gain in performance can be obtained when the middleware
adapts the distribution policy to the actual needs of applications.

Third, and this is a unique feature among distributed shared data space systems, the framework adjusts the distribution policy used for tuple types to the usage pattern of applications during execution. This makes it suitable for dynamic applications that change behavior during runtime. Our adaptation mechanism is based on models to predict cost values for system parameters (latency, bandwidth use and memory use).

Finally, we provided a mechanism by which these models are calibrated automatically. The automatic calibration alleviates the burden on the system designer by avoiding the need for obtaining detailed measurements about the environment in which the application will be deployed. The comparison between the predicted values and the measured values show that a reasonable accuracy of these models is obtained.
Chapter 4

Availability

An earlier version of this chapter has appeared as: G. Russello, M. Chaudron, M. van Steen, Dynamically Adapting Tuple Replication for High Availability in a Shared Data Space, In Jean-Marie Jacquet and Gian Pietro Picco eds., Proc. 7th Int’l Conf. on Coordination Models and Languages (Coordination 2005), volume 3454 of Lecture Notes in Computer Science, Springer-Verlag, pages 109-124, Namur, Belgium, April 2005.

An extended version of this chapter has been accepted for publication as: G. Russello, M. Chaudron, M. van Steen, I. Bokharouss, Dynamically Adapting Tuple Replication for Managing Availability in a Shared Data Space, special issue for Science of Computer Programming.
4.1 Introduction

So far, we have considered separation of concerns and adaptation for performance, focusing on metrics such as application-perceived latency and consumed network bandwidth. In this chapter, we concentrate on availability. Assuming that components may unpredictably fail, particular care has to be taken for shared data items to remain available to other components. Similar issues arise when an application is deployed on mobile nodes. In such an environment, a node’s connectivity may be highly unpredictable and a set of data items may unexpectedly disappear when a node disconnects.

A well-known solution to deal with data availability is replication. By replicating data on several nodes, the system can statistically guarantee that a data item is available even if the node where the item was inserted is no longer connected (or has failed). However, replicating for availability may conflict with replicating for performance. For example, high performance requirements may dictate that only weak data consistency can be supported, whereas high availability generally requires that updates are carried out on all replicas simultaneously.

Such tradeoffs generally require application-specific solutions. However, instead of imposing a single solution, we propose a framework that offers to the application developer a suite of replication policies. Each policy incurs costs with respect to performance, availability, consistency, etc. In our approach, a developer is offered a simple means to weigh these different costs such that the system can automatically choose the policy that meets the various (and often conflicting) objectives best. Moreover, through continuous monitoring of the environment the system can dynamically and automatically switch to another policy if it turns out that this would reduce overall costs.

We make the following contributions. First, we provide a simple mechanism that allows for separating concerns regarding performance and availability in shared data space systems. Second, we demonstrate how possibly conflicting objectives can be dealt with in these systems, such that the selection of a best policy can be done dynamically and in a fully automated fashion. Third, we show that the input needed from an application developer to support these optimal adaptations can be kept to a minimum, allowing the developer to concentrate on the design and implementation of functionality.

This chapter is organized as follows. In Section 4.2 we present the extensions of GSpace’s architecture for dealing with availability. To prove the soundness of our framework we conducted some experiments, of which the outcomes are discussed in Section 4.3. Section 4.4 discuss related work. We conclude in Section 4.5.
4.2 Architectural Design for Availability

In this section, we discuss the changes introduced in the architecture of GSpace for dealing with availability. Figure 4.1 shows the internal structure of a GSpace kernel deployed on a networked node. A GSpace kernel consists of two subsystems: the Operation Processing Subsystem (OPS) and the Adaptation Subsystem (AS).

The internal structure of the OPS is described in detail in Chapter 3. Here we concentrate on the structure of the AS since it is the submodule that is affected by the introduction of the availability concern.

The AS is an optional addition to GSpace that provides the functionality needed for dynamic adaptation of policies. The AS communicates with the co-deployed OPS for obtaining information about the status and actual usage of the system. In particular the Logger is responsible for logging all the space operations executed on the local kernel. When the number of operations for a particular type reaches a threshold, the logger notifies its local Adaptation Module (AM). The AM is the core of each AS. The AM coordinates the different phases of the adaptation mechanism. The code of the AMs on all nodes is identical. However, for each tuple type in the system one AM operates as a master and all the others as slaves. The master AM decides which replication policy should be applied to a tuple type. The slaves AMs follow the master’s decisions. The Cost Computation Module (CCM) and Replication Policy Cost Models (RPCM) are responsible for computing the costs incurred by the replication policies for a given set of operation logs. The Transition Policy prescribes how to handle legacy tuples in order for them to be
placed at locations where the new replication policy expects to find them. The *Adapt-Comm Module (ACM)* provides communication channels between the ASes on different nodes in the system.

The modules that were added for dealing with availability are the following:

**Availability Sensor:** This module is responsible for measuring the availability of the node in which it is deployed. This is done by periodically writing timestamps in a file. When a failure occurs, this timestamp is used to compute the duration that the node was not available.

**Group Generator:** Generating groups of nodes is the task of this module. Once the availability values for all the nodes have been collected, the master AM passes this information to its local Group Generator. The Group Generator will aggregate nodes following some given strategy. For instance, in the experiments that we discuss in Section 4.3 the Group Generator selects the best 3 nodes in term of availability. The generated group is then passed to the replication policies.

In the following section we describe in more detail how the different modules in the AS contribute to the mechanism that allows GSpace to select the replication policy that best suits the application behavior.

### 4.2.1 Autonomic Behavior in GSpace

This section describes the mechanism that allows GSpace to dynamically evaluate and select the availability policy that fits best the needs of the application.

In a distributed system such as GSpace, tuples are often stored and accessed remotely. Since nodes may fail or get disconnected, part of the shared data space could not be reachable. A common solution to this problem is the use of replication. By replicating tuples across several nodes we increase the probability of accessing a tuple even if some nodes are down. However, replication requires consumption of extra resources, such as extra memory for storing tuple replicas and bandwidth for exchanging information needed for keeping the replicas in a consistent state. Also, keeping replicas consistent comes at the price of global synchronization when updates occur.

Instead of proposing a one-size-fits-all solution, our approach sets flexibility as its primary goal. We included in GSpace a suite of replication policies each with its own tradeoff between provided availability, resource consumption, and performance. In this chapter, we ignore performance issues, allowing the application developer to specify only the availability requirements for the tuple types used by the application. The problem is now shifted to finding the replication policy that minimizes resource consumption while fulfilling the availability requirements.
These conditions are generally in conflict with each other. As we will show, our simple mechanism is able to deal with such conflicting situations in a fully automated fashion.

As the environment’s conditions change over time, a static assignment of replication policy to tuple type could eventually fail to provide the required performance of the system. As a solution to this issue, we monitor the environment. Application patterns are detected by logging each data space operation. Moreover, to guarantee that availability requirements are fulfilled, sensors are placed in each node to measure node availability in real time. By combining these data, our mechanism can automatically detect when to switch to another replication policy if it turns out that availability is at risk, or when resource consumption can be improved.

We identify three phases in our mechanism, which we explain in turn.

- monitoring phase
- evaluation phase
- adaptation phase

**Monitoring Phase**

During the first phase GSpace collects statistical data regarding its environment. This data consists of information about the availability of nodes and the usage profile of application components.

For collecting information on node availability, the GSpace kernel is instrumented with a sensor that monitors the availability of the node where it is running. Before diving into implementation details, we introduce the basic math behind the measurements that our system performs. The formula for calculating the availability of a single node is:

\[
\text{Availability} = \frac{\text{Mean Time To Failure}}{\text{Mean Time To Failure} + \text{Mean Time To Recover}}
\] (4.1)

It is important to understand what exactly Mean Time To Failure (MTTF) and Mean Time To Recover (MTTR) mean. With MTTF we indicate the average time that the node is continuously operating, i.e. the average time between the end of one failure and the beginning of the next. With MTTR we address the average time necessary for the node to recover from an experienced failure.

Figure 4.2 sketches the time line of a node that experiences some failures. When a failure \( i \) occurs we indicate with \( s_i \) and \( e_i \) respectively the time when the failure \( i \) starts and ends. We assume that the starting time of the node (the very first
time that the node is activated) is equivalent to $e_0$ (end of failure 0). Figure 4.2 also provides a graphical representation of MTTF and MTTR to understand how to compute those values. For instance, the availability value after the $n$-th failure is obtained by the following formula:

$$\text{Availability} = \frac{\sum_{i=1}^{n}(s_i - e_{i-1})}{\sum_{i=1}^{n}(s_i - e_{i-1}) + \sum_{i=1}^{n}(e_i - s_i)}$$

(4.2)

For computing (4.2) we need to collect the starting and ending times of all failures. When the system is started for the first time, the sensor writes into a file the starting time of the system. Periodically (every 100ms), the sensor is activated and writes timestamps into the same file. Writes to disk are synchronous. Actually, a timestamp is just the time at which the sensor is active. After a node experiences a failure, at re-booting time the sensor detects that the system was down (since the timestamp file is stored persistently). The starting time of a failure then is considered as the time at which the last timestamp was written whereas the time at which the system is up again is considered as the end-of-failure time. GSpace simply calculates the downtime as the difference between the new starting time and the time of the last executed timestamp.

For collecting information about the application behavior, we employ the same method as described in Chapter 3. Each data-space operation that application components execute is logged and stored per tuple type. Figure 4.3 shows the message sequence chart that describes the operation logging. The data that is logged contains:

- Operation type: the space operation executed (either a read, take or put)
- Tuple type: the type of the tuple or template passed as argument with the operation
- Location: the address of the GSpace kernel (i.e., node) where the operation is executed

![Figure 4.2](image-url) The time line of a node that experiences some failures.
4.2 Architectural Design for Availability

- Tuple ID: a unique id provided to each tuple that enters the shared data space
- Tuple size: the size of the tuple inserted through a put operation or returned by a read or take operation
- Template size: the size of the template passed as argument of a read or a take operation
- Timestamp: the time when the operation is executed

When the number of executed operations on a node reaches a given threshold the system starts the next phase.

Evaluation Phase

The evaluation phase consists of collecting data from all nodes and comparing the cost of different replication policies.

Figure 4.4 shows the message sequence chart of the evaluation phase. The master AM requests all slave AMs to send their local data (logs and node availability). This data is combined and the costs for each policy are calculated by means of simulation.

For capturing the performance of the different replication policies we use a cost function. Our cost function is a linear combination of various parameters. The values of these parameters are combined in an abstract value that quantifies the tradeoff between performance versus resource usage for a given replication policy. The parameters are defined in such a way that a lower value indicates lower costs (and thus better behavior). The replication policy that leads to the lowest costs is the best policy for the application.

![Figure 4.3](image-url) The MSC of operation logging.
In this work, we apply the same method as described in Chapter 3 with the focus on data availability. Therefore, we use a different cost function: $bu$ represents the bandwidth usage; $mu$ represents the accumulative memory usage; and $da$ represents the derived availability. The latter is calculated as follows:

$$
da_p = \begin{cases} 
100 - \text{availability}(p) & \text{if } \text{availability}(p) \geq \text{required availability}, \\
\text{MaxValue} & \text{if } \text{availability}(p) < \text{required availability}
\end{cases}$$

In this way, if the availability provided by a replication policy $p$ does not satisfy the user’s requirements then the value for $da$ is set to MaxValue (with MaxValue $\geq 100$) so that the calculated costs will become very high and the system will automatically reject this policy. The cost function is defined as follows:

$$CF(p) = w_1 * bu_p + w_2 * mu_p + w_3 * da_p$$  \hspace{1cm} (4.3)$$

The weights $w_i$ tune the relative contribution of each parameter to the overall cost.

Once the costs are calculated for each replication policy, they are passed to the AM that selects the best replication policy. The AM checks whether the current policy is still the best one. If this is the case, no further actions are undertaken. Otherwise, the AM starts the phase described next.
Adaptation Phase

In this phase the system switches replication policy and adapts the data-space content. In Figure 4.5 the actions executed during this phase are presented in a message sequence chart. The master AM freezes application operations for the given tuple type in all nodes. The master AM set the best policy as the actual policy by updating its local policy table. Afterwards, it commands all slaves to update their local policy table, setting the best policy as the policy for the tuple type.

At this point, the master AM resumes the operations in all nodes. Here, the adaptation phase for availability differs from the adaptation phase for performance, described in Chapter 3. In that context, before the end of the adaptation phase, each AM uses a transition policy associated with the tuple type for dealing with possible legacy tuples present in the local data-space. For this concern, the replication policy takes care by itself of those tuples. In fact, all replication policies are based on our version of the group communication protocol described in [39]. This version of the protocol takes care of the legacy tuples replicated across the group of nodes. For instance, if a node is added to the group, the protocol makes sure that all the tuples replicated on the old nodes of the group are inserted in the new node.
4.3 Implementation and Experiments

This section describes the experiments that we performed using a full-scale GSpace setup deployed together with a benchmarking application.

In Chapter 3 experiments focused on distributed systems in which application components dynamically join and leave a system during execution (but in which the nodes were always available). In that chapter we showed that there is no single distribution policy that is best for this dynamic type of application behavior. Furthermore, we showed that dynamically adapting the distribution policy outperforms any static policy.

In this chapter we do not only consider changes in the application behavior, but also in the underlying hardware infrastructure. In particular, we consider that the availability characteristic of nodes in the network may change. This occurs, for instance, in ad-hoc networks where devices join and leave a network.

We fully implemented all modules to enable a GSpace kernel to be availability aware. The experiments were executed in a cluster of nodes connected by a gigabit LAN. We deployed application components and GSpace kernels in several nodes (ranging from 3 up to 17) of the cluster. While application components were executing we performed controlled failures on the nodes. The goals of this set of experiments are:

- Firstly, we want to show that without adaptation, no single static policy is able to sustain a given level of availability. Moreover, we want to show that the dynamic adaptation of the policy provides a better level of availability in the case of changing infrastructure.
- Secondly, we want to show that the system is able to dynamically adapt to the application behavior changes as well as to the infrastructure changes.
- Thirdly, we want to measure timing properties of the Adaptation Subsystem under several circumstances. In particular, we want to measure the tradeoff between the overhead and the gain that the system obtains when adaptation is enabled.
- Finally, we are interested in how well the system scales in terms of frequency of evaluation and number of nodes.

In the following, we first describe the setup used for the experiments.

4.3.1 Setup of the Experiments

We run the experiments on the Sandpit cluster hosted by the Eindhoven University of Technology. The nodes used for the execution of the experiments are 3.06 GHz Pentium4 with 2 GByte memory and are connected via a 1 Gigabit LAN.
For benchmarking our system, we used an application benchmark that we used in Chapter 3. Figure 4.6 shows the deployment of such an application together with GSpace kernels. In each node, an application component is deployed together with a GSpace kernel.

The core unit of the application is the **Coordinator**. The main functions of the Coordinator are the following:

- generate sequences of operations, that we refer to as *runs*, according to a predefined application behavior.
- generate sequences of failures according to a specific pattern (per node).
- the ability of re-playing exactly the same run of operation.

The Coordinator avoids the introduction of randomized anomalies in our experiments. The Coordinator orchestrates the execution of a run by dispatching operations to components according to a given sequence. The Coordinator is connected directly with each application component and does not interact with any GSpace kernels. Thus, Coordinator-Component interaction does not affect any decision taken by the Adaptation Subsystem.

For this set of experiments we also want to control availability of nodes over time. More precisely, we want to control when and for how long a failure should occur on each node\(^1\). For this reason, we extended the runs by adding a set of special operations for starting and ending a kernel’s failure.

\(^1\)A failure means that a kernel does not respond to a request.
4.3.2 Replication Policies

The set of replication policies for GSpace is extensible. For the experiments in this chapter, we use the following set of replication policies:

- **Full Replication (FIR).** This policy puts a copy of every tuple on every node in the system (as soon as a tuple is inserted).

- **n-Fixed Replication (n-FxR).** This policy replicates tuples to a group of *n* fixed nodes (as soon as the tuple is inserted). When awareness of node availability is enabled, the Availability Manager provides the *n* best nodes where tuples should be replicated.

- **Dynamic Consumer Replication (DCR).** This policy replicates tuples to all nodes that host an application component that is a consumer of this type of tuple. In case the availability of the consumer group cannot provide the required availability, nodes are added following the decision of the Group Generator.

- **Dynamic Producer Replication (DPR).** This policy replicates tuples to all nodes that host an application component that is a producer of this type of tuple. Nodes provided by the Group Generator might be included in the group of producer nodes whenever this group cannot sustain the required availability.

For maintaining consistency among the nodes where replicas are stored, the replication policies collaborate using a group communication protocol [39]. The nodes on which tuples are replicated are joined in a group where the operations are executed atomically. Moreover, the group communication protocol takes care of consistency issues that could arise from the failure of some of the nodes in the group.

The availability under a given replication policy is determined by the availability of the group of nodes where tuples are replicated. In particular, a group of nodes is considered available if at least one node of the group is available. Then, the group availability for a specific tuple type, $GA$, equals 1 minus the probability that all nodes within the group fail:

$$GA = 1 - P_{\text{all nodes down}}$$ (4.4)

We assume that failures of nodes are independent. Then, the probability that all nodes fail is equal to the product of the probabilities of failure $f_i$ of the individual nodes:

$$P_{\text{all nodes down}} = \prod_{i=1}^{n} f_i$$ (4.5)
4.3 Implementation and Experiments

Figure 4.7 Memory Usage measured for the different replication policies.

4.3.3 Adding Awareness of Node Availability to Policies

The experiments in this section show that by constantly monitoring the underlying infrastructure, the GSpace system improves sustainability of the required availability requirements despite the unpredictable behavior of the nodes.

For the experiments in this section 10 nodes were used. We fixed the application behavior according to the following pattern. All the application components act both as consumers and producers.

For this application behavior, both DCR and DPR policies replicate the tuples in all nodes. This means that the memory usage is the same as that for the FIR policy, as Figure 4.7 shows. Instead, the memory footprint of the 3-FxR policy is smaller than that of the other policies since this policy replicates tuples on a smaller number of nodes.

First we consider the case when the availability monitoring is disabled. The required availability for the tuple type used in the experiments is 70%. The 3-FxR policy is defined to use the three nodes that provide the highest availability at the moment the system is started. However, the availability behavior of these nodes is programmed to decrease from 90 to 10 in steps of 5 (percent).

Using these three nodes the 3-FxR policy initially satisfies the availability threshold. However, during execution, the nodes that are used by the 3-FxR policy experience an increasing number of failures. Hence, the availability of the nodes decreases and as a result, the availability that the 3-FxR policy provides decreases.
Figure 4.8 Measured availability for the replication policies when availability awareness is disabled.

Figure 4.9 Cost Function for the replication policies when availability awareness is disabled.
4.3 Implementation and Experiments

Figure 4.10 Measured availability for the replication policies when availability awareness is enabled.

Figure 4.8 clearly shows this decreasing behavior. The other replication policies provide a fairly stable availability with minor fluctuations. This is due to the fact that the changing availability of three nodes out of 10 has a minor impact on their overall availability.

The previous graphs were concerned with availability. Next, we look at the effect of the replication policies on the cost function.

From Figure 4.9, we can conclude that as long as the availability requirements are met, 3-FxR is the best policy since it uses the least memory. However, around the 10th execution cycle this policy can no longer sustain the required level of availability. As a result, the cost function value increases dramatically.

Next, we re-execute the same sequence of operations but this time with the availability monitoring enabled. The 3-FxR policy still makes only a fixed number of copies, but now it selects the three nodes that have the highest availability at the time of evaluation\(^2\).

The memory usage graph is the same as the one shown in Figure 4.7 since the application behavior is the same. However, now the system is able to select nodes based on the measured availability of the nodes. At each evaluation, the system selects the three nodes that have highest availability. Figure 4.10 shows that 3-FxR is able to provide the required availability. Moreover, since the memory footprint

\(^2\)These nodes are provided by the Group Generator module
is lower than that of the other policies, 3-FxR is always the best policy. This is shown in the cost function graph on Figure 4.11.

4.3.4 Dealing with a Changing Environment

In this section, we discuss the results obtained when both application behavior and node availability change during execution of the system.

The results will show that our mechanism not only is able to select the replication policy that satisfies the availability requirements but also it selects the policy that best suits the components’ behavior.

During these experiments the availability characteristics of nodes are measured from the system and made available to GSpace. The application component behavior is programmed to change during execution according to the following phases:

- Phase 1 (execution cycles 0–32): all application components act both as consumers and producers;
- Phase 2 (execution cycle 32–64): only the application components deployed on nodes $n_9$ and $n_{10}$ act as consumers, all the other components act as producers;

Figure 4.11 Cost function for the replication policies when availability awareness is enabled.
4.3 Implementation and Experiments

- Phase 3 (execution cycle 64–95): only application components on nodes $n_9$ and $n_{10}$ act as producers, the other components act as consumers.

Additionally, the availability of nodes $n_9$ and $n_{10}$ is programmed to oscillate between 10% and 90% during the execution of the entire experiments. Therefore, the group formed by these two nodes is not always able to sustain the required level of availability, which is fixed to 70%.

Let us begin with analyzing the cost function values on Figure 4.12. During the first phase of the execution, the best policy that can guarantee the availability requirements with minimal memory usage is 3-FxR.

During the second phase of execution, DCR is the best policy. This is due to two factors. Firstly, only two nodes host application components that act as consumers. Therefore, DCR uses a group of nodes that is at most as large as the group used by 3-FxR. This has a major impact on the memory usage, as Figure 4.13 shows between execution cycles 32 and 64. In fact, when the combined availability of node $n_9$ and $n_{10}$ is above the required availability, Dynamic Consumer Replication has a smaller memory usage footprint than 3-FxR. However, sometimes those two nodes are not enough to guarantee the required availability. Thus, DCR has to include other nodes to sustain the required availability. This is done by adding a node that is selected by the Group Generator module. Secondly, DCR incurs in a reduced bandwidth usage as is shown in Figure 4.14.

The last phase of execution witnesses another change. Application components
Figure 4.13  Measured memory when the application behavior changes.

Figure 4.14  Bandwidth usage when the application behavior changes.
4.3 Implementation and Experiments

Figure 4.15  Availability values when the application behavior changes.

switch behavior. In particular, after evaluation cycle 64, the application components on nodes \( n_9 \) and \( n_{10} \) start acting as producers. All the other components start to act as consumers.

After a transition phase between cycles 64 and 70, where the components’ behavior stabilizes, the DPR becomes the best policy, as Figure 4.12 shows. This is mainly due to the same factors that we discussed for DCR. This is confirmed also by the graphs in Figure 4.13, after the stabilization phase.

To conclude, Figure 4.15 shows that in all cases the availability sustained by the policies used in the different phases is always greater than the required value. This is an improvement over the behavior that is oblivious to changes in availability of nodes, yet the adaptation happens transparently to the application.

In the following section, we will present results on the overhead and scalability of the Adaptation Subsystem.

4.3.5 Adaptation Mechanism Performance

We used a real deployment of the system together with a benchmark application to perform some performance measurements of the adaptation mechanism. Here, the application behavior pattern does not change over time. What changes is the availability of nodes.

The results that we obtained can be divided in three groups. The first group
Chapter 4 Availability

Figure 4.16 Read and take latency values different experimental settings.

concerns the application perceived latency. In particular, we analyzed the tradeoff between the overhead and the gain in terms of latency that application components perceived due to the execution of the adaptation mechanism.

The second group of results concerns the evaluation phase of the adaptation mechanism. In particular, we analyzed how the time used for evaluating is influenced by:

- The length of a series of operations (anywhere in the system) that triggers the evaluation phase. From now on we will refer to this length as $L_e$.
- The number of nodes used for conducting the experiments.

The last group of experiments focuses on the correctness of the decisions taken versus the specific overhead of the adaptation phase.

Overhead vs. Gain of the Adaptation Mechanism

In this section we present some measurements that we obtained for estimating the general overhead that the adaptation mechanism introduces.

For estimating the overhead, we measured the application-perceived latency in executing blocking operations, such as read and take. We used the same run of
10000 operations equally composed of reads, puts and takes. The run was executed over 10 nodes.

During the experiments, failures could occur simultaneously on several nodes. This means that all nodes in the group where tuples are replicated could be down at the same time. As a consequence, a read or a take operation could be blocked until one of the nodes in the group recovers from the failure and can serve the operation.

We executed the same run of operation using the following settings:

- net-latency no-adapt (nlna): in these settings, down time and recovery time of a failure were not included in the latency measurements of components executing read and take operations. Moreover, the Adaptation Subsystem was not activated.

- net-latency with adapt (nlwa): as in nlna, but this time the Adaptation Subsystem was activated.

- total-latency no-adapt (tlna): in this case, application latency was measured taking into account the down time and recovery time due to a failure. The Adaptation Subsystem was not activated.

- total latency with adapt (tlwa): as in tlna, but with the Adaptation Subsystem activated

During the experiment we measured for a component $c_i$ the values $\text{avg}_i \text{read}$ and $\text{avg}_i \text{take}$. Averaging again these values for the number of components we obtain the average time for read and take operations. The results in milliseconds are showed in Figure 4.16.

In Figure 4.16-(a) the measurements for nlna and nlwa are compared. There is a slight increment of latency in nlwa compared to the latency in nlna. This increment is due to the Adaptation Subsystem, and in particular when the adaptation phase is executed. In fact during this phase, all the operations in the system are frozen for avoiding inconsistency issues. This effects the latency time perceived by application components.

However, when down time and recovery time are taken into account things change drastically. Figure 4.16-(b) shows that when down time and recovery time are accounted, the average latency that a component perceives with adaptation is almost half of the time when adaptation is not activated. This is due to the adaptation that dynamically changes the group of nodes where the tuples are replicated taking into account the availability of each node. This is further confirmed by counting the numbers of group failures that occurred during the experiments. In fact, when adaptation was not enabled (tlna) the number of group failures accounts to 45. In the case of adaptation enabled (tlwa), only 19 group failures occurred.
In the next section we will concentrate on the influence that $L_e$ and the number of nodes where kernels are deployed have on the evaluation phase.

**Evaluation Phase Measurements**

In this section, we present some results on the performance of the evaluation phase when $L_e$ and the number of nodes change.

Each evaluation phase quantifies the past behavior of the system by simulating it. A simulation mimics the past system behavior using traces of the operations that were executed up until the moment of evaluation. The higher $L_e$ the longer the traces that are used for the evaluation. The traces are collected by each kernel in the system by logging the operations that it executes. When the evaluation is triggered, the logs from each kernel are collected at a single kernel (the Master AM) where the evaluation is executed. As a consequence, the higher the number of nodes the longer the Master AM takes for aggregating and assembling the logs from the each node. Our question is how well the evaluation execution time scales with respect to $L_e$ and the number of nodes.

To find this out, we executed two sets of experiments. For each set the execution time of each evaluation phase was measured. In the first experiment set, we executed the same run of 10000 operations on a 10-nodes setup for different values of $L_e$. Figure 4.17(a) shows the evaluation execution time in milliseconds (Y-axis) for different values of $L_e$. As the graph shows, the evaluation time does not scale linearly with respect to the threshold. Although $L_e$ increases 20 times (from 50 to 1000 of operations) the increment of the evaluation time is only by a factor 6.

In the second set of experiments, we used a run of 10000 operations and a fixed threshold value of 250. Here, we use a different number of nodes for each execution. Figure 4.17(b) shows the evaluation times obtained for the different numbers of nodes. The results show that although the number of nodes increases almost 6 times (from 3 to 17) the evaluation time increases by a factor of 2. The aggregation time is linearly dependent on the number of nodes. In fact, the Master AM sends point-to-point messages to the other kernels asking for their logs. However, the results show that the contribution of the aggregation time to the overall evaluation time is small compared to the execution of the simulation.

We conclude that the evaluation time scales well for a local-area network. For $L_e$ we used a quite wide spectrum of values. In fact, as we will see in the next section, a much smaller interval is required.

In the following section we will study the dependencies between $L_e$ and the percentage of correct decisions taken by the adaptation mechanism.
Figure 4.17 Average execution time for evaluation phase when (a) threshold and (b) number of nodes changes.
Figure 4.18  Node adaptation index (a) and execution penalty (b) for the adaptation phase.
4.3 Implementation and Experiments

Node Awareness versus Node Adaptation Penalty

In order to assess the efficiency and effect of our approach, we consider how $L_e$ influences the quality of evaluations. At each evaluation we look at the most recent $L_e$ operations and at the availability of nodes. Based on this a combination of an availability policy and a set of nodes is chosen that together provide a certain level of availability as well as minimize the costs of replication. If this combination differs from the current settings the policy or group of nodes will be changed.

In this section we focus on the part of the adaptation mechanism that makes a selection of nodes. We analyze the impact of the frequency of node evaluations on the performance of the adaptation mechanism.

The problem of determining an appropriate value for $L_e$ requires finding a proper balance between the following concerns:

1. Taking $L_e$ too large may cause the mechanism to fail to adapt quickly to changes in the node availability.
2. Taking $L_e$ too small may cause a performance penalty because evaluations are performed too often.

Firstly, we investigate how $L_e$ influences the percentage of adaptations. Secondly, we show how $L_e$ relates to the execution penalty of the adaptation mechanism.

For measuring the influence that $L_e$ has on the adaptation decisions we performed the following experiment. We executed a run of 10000 operations on a setup of 10 nodes using different values for $L_e$. During the execution only node availability is changed. The application behavior is maintained the same through the entire execution. In particular, all the application components behave as consumer and producer of tuples.

For each run, we measured the number of times that the evaluation phase was triggered and the number of times that such evaluation leads to an adaptation of the set of nodes. If an adaptation happens, this means that the previously selected nodes were no longer the best. Consequently, the fewer nodes we need to change, the better our mechanism was capable of keeping up with actual availability. We define the node adaptation index as:

\[
\text{node adaptation index} = \frac{\text{num node adaptations}}{\text{num evaluations}}. \tag{4.6}
\]

The lower this index, the better we were able to keep track of changes in node availability. The results of these experiments are shown in Figure 4.18(a). Not surprisingly, we can see that for smaller values of $L_e$ the system is better able to select the nodes with highest availability.
However, higher responsiveness comes at a price. For the aforementioned experiments, we measured the total execution time spent in changing nodes in a group (excluding evaluation time). The results of these measurements are shown in Figure 4.18(b)). The time values on the Y-axis are given in milliseconds. The graph clearly shows how small thresholds incur a higher penalty. During these adaptations all application requests are frozen to avoid the introduction of inconsistency. Hence this causes an increase in application-perceived latency.

### 4.4 Related Work

This section describes other approaches for increasing availability of shared data space.

PLinda [38] is a variant of Linda that addresses fault-tolerant applications. In PLinda both data and processes are resilient to failures. In particular, by using a transaction mechanism extended with a process checkpoint scheme, PLinda ensures that a computation is carried out despite node failures. Compared to our approach, PLinda offers more functionality since it is resilient against process failures. On the other hand, in PLinda application developers have to explicitly declare which part of their application code should be executed in a fault-tolerant fashion. Therefore, application code is interwoven with extra-functional concerns not relevant to the application functionality.

Another fault tolerance implementation of Linda is FT-Linda [3]. As for PLinda, FT-Linda supports a transaction mechanism that allows the recovery of data and processes after a failure. However, FT-Linda requires the application developers to put extra effort in making their application resilient to failures. For instance, the application developer has to program the application to take care of removing intermediate results after a failure. Again, this violates separating different concerns in the application design.

Although it was designed for taking advantage of idle time of workstations for running parallel applications, Piranha [40] could be used for addressing fault-tolerant applications as well. In Piranha, worker processes execute tasks on idle workstations. As soon as a workstation becomes busy, a worker process has to stop its current computation. The task has to be carried out by another Piranha worker on another idle workstation. Therefore, a retreat has the same effect as a failure. The Piranha model assumes that the execution of the task is carried out atomically despite the retreat. As for the FT-Linda, the Piranha system requires the application developer to program the application to clean-up intermediate results when a task has to retreat. Again, we see that application code is interwoven with fault-tolerant concerns.

An alternative approach to transaction mechanism for building shared tuple spaces is proposed in [69]. In this work, Rowstron proposes exploiting code mobility as a
4.4 Related Work

mechanism for fault tolerance. By using code mobility, the system can guarantee an operational semantics in which either all operations are executed or none. The approach uses a run-time system that contains a checkpointing mechanism. In this way, the application developer does not need to interweave fault-tolerance code in an application since the runtime system will deal with this. To address the removal of legacy data left by mobile agents that are no longer alive, the author introduces the notion of agent wills. The agent-will is a small piece of code embedded with the runtime system that describes what to do with data after the agent ceases activity. This will-code is executed by the runtime system whenever it detects that the respective agent crashed.

An evaluation of fault-tolerance methods for large-scale distributed shared data spaces is described in [87]. The authors divide the methods for providing fault tolerance in two groups: run-time level and application level. The first group contains methods that provide fault tolerance only for the shared data space content. These methods make sure that the space content is not lost even if one or more nodes fail. The second group contains methods that extend fault-tolerant guarantees to the application level. These methods can ensure that a sensitive tuple (such as one acting as a lock) is not lost in case the application component that retrieved it fails. Clearly GSpace falls in the first group.

The authors continue their evaluation by describing the mechanisms that are typically used for fault tolerance. These include transactions, mobile code, checkpointing, and replication. The set of availability policies that can be built for GSpace utilize checkpointing and replication as basic mechanisms for providing fault tolerance.

Interesting enough, the authors conclude their evaluation pointing out that for supporting fault tolerance in Linda-like systems it is desirable to separate the set of primitives in two levels: functional and non-functional. Of the two, the latter should be used for specifying actions related to fault-tolerant concern. This is exactly in line with our way of dealing with fault tolerance, although we go further by completely separating this concern from the application functionality. Another interesting conclusion of the authors is that supporting run-time adaptation of fault-tolerant mechanisms can improve the optimization of the resource usage that fault-tolerant mechanisms require. This is exactly what the experimental findings of our research show.

An alternative approach is proposed in SwarmLinda [50, 88] in which a randomized algorithm based on swarm intelligence is used to direct searches to tuples. In this case, we see that scalability may be achieved so long as the search algorithms prove to be efficient. Note, however, that SwarmLinda by itself does not explicitly address fault tolerance, although by its nature allows a shared data space as a whole to adapt to process failures. In such cases, a query will just follow a different route to a matching tuple.
This adaptive nature is also explored in the TOTA middleware [47, 48] where tuples are equipped with a piece of code telling them how they should move through a distributed data space. In TOTA, tuples are not associated with individual data spaces or nodes, but instead are directly inserted into an overlay network, from which point on they can be propagated and diffused according to their own rules. Moreover, during this process, their content is allowed to change making it possible to adjust to the context in which a tuple currently resides. Again, it is the flexibility of the approach that will allow handling of failures, although fault tolerance by itself is not explicitly dealt with in the work on TOTA described so far.

4.5 Conclusions

In this chapter we made the following contributions. First, we provided a simple mechanism that allows for addressing availability concerns in shared data space systems separately from the functionality of applications. As a result, different policies can be employed for achieving different availability characteristics without affecting the functionality of the application.

Second, we demonstrated how possibly conflicting objectives (such as high availability and low resource use) can be dealt with in a fully automated fashion through the use of a cost-function.

Third, we showed that the input needed from an application developer to support these optimal adaptations can be kept to a minimum, allowing the developer to concentrate on the design and implementation of functionality.

Finally, we showed the superior performance of dynamically adapting the replication policy that is used. The experiments showed that our mechanism is able to dynamically adapt the replication policy to the availability characteristics of the infrastructure. Moreover, the mechanism takes into consideration the application behavior and selects the policy that suits best the application needs.
Chapter 5

Security

“Dottori, difficilissimo è”.
“Perché?” “Perché c’è la guardia ai passi”.
Montalbano strammó. Quale guardia? Quali passi?
“Cataré, che minchia dici?” “Dottori, ora ci lo spiego. Quando uno non voli che uno talia le cose intime che ci ha dintra, ci mette una guardia ai passi”.
Montalbano accapí. “Una password”.
“E io che dissi? La stessa cosa dissi. E se uno non ci dice la parola d’ordine, la guardia ai passi non ti fa passare”.
“Allora siamo fottuti?”

Andrea Camilleri, “La luna di carta”.

5.1 Introduction

The SDS model was conceived as a coordination medium for easing the task of coordinating the computation of distributed parallel programs. The first implementations were closed systems, in the sense that they were realized by compiling application and SDS code altogether. Once the system is deployed and executed, it is not possible to add or remove application components. Thus, in such a system security is not an issue.

However, closed systems offer limited flexibility for distributed applications. In contrast with closed systems, open systems were introduced where the SDS was not bound to an application but it was an autonomous application by itself. The main advantage of this type of systems is that persistent data storage can be offered to applications. In this way, application can dynamically join and leave the computational environment. This is clearly a feature that appeals distributed applications. These systems were deployed in LANs, to which access was limited to a fixed community strictly controlled by system administrators. Because the programs were running in such a closed environment security was still a manageable risk.

The last decade has witnessed a rise of interest by software engineers towards the SDS model. Applications are becoming increasingly distributed and a simple means for collaboration is a must. With its small API and the decoupling of communication in space and time, the SDS model has been exploited as a coordination layer for distributed applications. However, these applications are deployed in open environments, such as the Internet. The entities that might access an SDS are more difficult to trace and control by system administrators alone. This is even more critical when the nature of the data that a tuple represents is sensitive. For instance, an e-banking application could be built using an SDS. In such a system, a tuple could be used for representing the balance of users. It is desirable that such tuples can be accessed by the respective users. However, this access should be restricted only to read operations to avoid that a user could maliciously or inadvertently changes it.

If SDSes are to be used successfully in such applications then security should be embedded in the SDS itself. Such a mechanism should be used for enforcing security policies, such as authentication, access control, and encryption.

For each of those policies, there may be several mechanisms for enforcing it. For instance, the process of authenticating a user can be carried out by means of a mere user id and password. Another way could be to use the user’s fingerprint. Still, another mechanism could be based on certificates signed by an authentication authority. Each mechanism might be specific for a given application. For instance, user id and password, and fingerprint mechanisms are well suited for use with humans, while a certificate could be more appropriate for authenticating a software artifact. For the sake of flexibility, it would be advisable that the security layer of
a SDS should be able to support several mechanisms for each security policy.

The aim of this chapter is to incorporate security as a separate extra-functional concern in GSpace, our SDS implementation.

The rest of this chapter is organized as follows. In Section 5.2 we provide a threat model from which our SDS should be protected. Additionally, we list several security properties with the threats that are prevented by enforcing such a policy. Section 5.3 provides a description of the GSpace architecture with its extensions for the security concern. How security settings are managed is described in Section 5.4. For validating our system we built an application where the full potentiality of our system is explored. The description of such an application is given in Section 5.5. In Section 5.6 we survey other approaches for integrating security mechanisms in SDSes is provided. We conclude with Section 5.7.

5.2 Security Policies and Threats

In this section we list several security properties. This is not intended to be an exhaustive survey, as this is not the scope of this chapter. Instead, we want to bring to attention some of the most important aspects related to security. For each security property, we consider a related threat model and some mechanism(s) for managing that property. The summary together with a short description for each property is shown in Table 5.1.

We aim at designing an architecture in which the application developer can concentrate on the basic functionality of the application, treating security as an orthogonal concern. The application developer should then just specify, separately from the application code, which security properties that he would like to apply in his application. Alternatively, if the threat-model view is offered, the application developer could specify which threats he would like to protect his application against.

To this end, we would like our architecture to support several security mechanisms that cover the set of properties presented in Table 5.1. Based on the developer’s specification, the system automatically assembles a security policy that satisfies those requirements. The assembling consists of activating inside the architecture components that implement the mechanisms described in Table 5.1.

To make things more concrete, let us consider the following example. The application developer wants to protect his application against masquerading and eavesdropping. When the system is deployed, the following mechanisms should be activated:

- An authentication mechanism that uses user id and password to verify the authenticity of the user. This mechanism is activated as soon as an applica-
<table>
<thead>
<tr>
<th>Property</th>
<th>Threat</th>
<th>Mechanism</th>
<th>Synopsis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Confidentiality</td>
<td>Comm. Eavesdrop</td>
<td>Encryption</td>
<td>Avoiding that the data is disclosed during communication</td>
</tr>
<tr>
<td>Confidentiality</td>
<td>Stealing Device</td>
<td>Encryption, key management</td>
<td>Avoiding that the loss of the device compromises the data</td>
</tr>
<tr>
<td>Integrity</td>
<td>Tampering</td>
<td>Redundancy, Checksum</td>
<td>Preventing that the data is maliciously modified</td>
</tr>
<tr>
<td>Accountability</td>
<td>Denial</td>
<td>Logging</td>
<td>Enforcing trace-ability of actions</td>
</tr>
<tr>
<td>Non-repudiation</td>
<td>Denial</td>
<td>Digital signature</td>
<td>Preventing an entity from denying the authenticity of its signature</td>
</tr>
<tr>
<td>Availability</td>
<td>Denial of Service</td>
<td>Accounting detection containment</td>
<td>Avoiding that a service is unavailable to authorized users/applications</td>
</tr>
<tr>
<td>Integrity</td>
<td>Replay</td>
<td>Time-stamping</td>
<td>Sending multiple copies of the same message should be avoided (reply attack)</td>
</tr>
<tr>
<td>Access Control</td>
<td>Masquerading</td>
<td>Authentication and Authorization, Identification</td>
<td>Avoiding that unauthorized entities access the data</td>
</tr>
<tr>
<td>Integrity</td>
<td>Concurrent Manipulation</td>
<td>Single Owner with pwd</td>
<td>Only one single manager of the system</td>
</tr>
</tbody>
</table>

Table 5.1 Security properties, threats and enforcing mechanisms.
5.3 Architectural Design for Security

In this section, we discuss the extensions of GSpace’s architecture for dealing with security concerns. We start describing the threat model that we take into consid-

![Figure 5.1 Threat points in a GSpace setup.](image)

- An authorization mechanism checks that the operation that is currently executed is part of the set of operations that the user/application is authorized to execute. This mechanism is activated when the operation request just arrived in the kernel but after the user/application has been authenticated.

- An encryption mechanism that encrypts/decrypts each tuple stored in the local slice, or that is sent to another kernel.

From this example we can observe that not all mechanisms should be activated at the same time. The order in which the security mechanisms should be activated should be coordinated according to the stage of execution of an operation. Thus, for the activation of the correct mechanism it is crucial that the architecture can monitor at which stage the execution flow of an operation is. We will come back to this issue in the next section where we discuss the extensions in the GSpace architecture.

5.3 Architectural Design for Security

In this section, we discuss the extensions of GSpace’s architecture for dealing with security concerns. We start describing the threat model that we take into consid-
eration, pinpointing where the attacks could occur in a typical GSpace setup. For each point that is prone to an attack, we identify the modules in the architecture where specific security mechanism should be inserted to counter the attack. Once these modules are identified, we describe how the architecture can be extended to accommodate modules that deal with security. Finally, we provide an example of a security policy and the details of how this policy is enforced at runtime.

5.3.1 Threat Model

As a first step towards the design of the architecture extensions let us determine which and where attacks could be executed in the system. Figure 5.1 sketches a typical GSpace setup with several application components, kernels and devices. The arrows indicate the points that are sensitive to security threats. Arrow (a) indicates the level where interaction between GSpace kernels and applications takes place. The security threats for this point are denial of service, replay and masquerading. Arrow (b) refers to the level where storage of the content in the kernel and device is handled. The threat is represented by the loss of confidentiality in case the device is compromised. The last level indicated by arrow (c) refers to the communication level between GSpace kernels. Here security threats are: communication eavesdropping, tampering, denial of service, and replay.

Those threats could be neutralized if appropriate security mechanisms are activated. The mechanism should be activated when the execution flow reaches specific modules in the architecture. For this reason, it is necessary to identify which modules are affected by a security threat to have the proper mechanism activated on time. For instance, an application that wants to execute an operation should be authenticated as soon as the request arrives in the module of a kernel that serves requests. A tuple should be encrypted whenever it is sent to another kernel via the module that deals with communication. Yet, a tuple should also be encrypted when it is stored in the local data space slice. In the next section we will describe which modules are added to the architecture and how security mechanisms could be activated.

5.3.2 Security Modules

Our aim is to extend the GSpace architecture for accommodating modules that deal with security. As we saw in the previous section, security could affect several of the modules already present in our architecture. Such modules should be somehow made security aware. In our current architecture, the modules that are affected by security are the following:

- The Controller, which is responsible for handling the communication between application and kernel. For instance, authentication and/or authorization
mechanism should be decided when the execution flow reaches this module.

- The Data Space Slice, which is responsible for (permanent and/or volatile) storage. Encryption for storage should be executed when the flow reaches this module.

- The Communication Module, which handles the communication between kernels. Data integrity and encryption for instance should be done when the execution flow enters this module.

To maintain the separation of concerns, we do not wish to modify modules of the existing architecture. This not only avoids a tedious and error-prone code modification but also ensures that our architecture is not entangled with any static security policy. This allows us to increase the flexibility of the architecture since several security policies can be applied, customizing the security policy to the application needs. Moreover, now it becomes possible to cope with dynamic changes in the security requirements of an application. A security policy can be changed at runtime without the need to recompile.

This can be achieved only if the architecture is such that:

- It is possible to detect when the execution flow reaches the sensitive modules,
- It contains separate modules for dealing with the decision-making process about security matters,
• It supports an extendable suite of security mechanisms.

Figure 5.2 shows the architecture with core modules and the new modules to deal with security. These new modules introduced are shown in Figure 5.2 in light gray. A description of such modules follows:

**Sec-wrapper** is responsible for making the module it wraps security-aware. The use of a wrapper avoids modifying the modules already present in the architecture. The tasks that a Sec-wrapper can fulfill could vary depending on the module that it wraps. A task common to all Sec-Wrappers is to notify the local Sec-Manager when the execution flow reaches its wrapped module. Other tasks include providing an interface between the wrapped module and the security mechanism that should be activated. Finally, a Sec-wrapper can execute special code that is given by a security mechanism (more on this later).

**Sec-manager** is the module responsible for selecting and activating the appropriate security mechanism. The selection of the mechanism is based on which Sec-wrapper notified the Sec-manager and for which tuple type the notification is sent. The type of mechanism that is selected is related to the context of the event. For instance, if the event comes from the Sec-wrapper of the Controller then the Sec-manager knows that a mechanism for authenticating the source of the request should be activated. However, several authentication mechanism could be available. The mechanism that should be activated is the one associated with the tuple type of the request. This information is stored in the Sec-table.

**Sec-table** associates tuple types with specific security mechanisms.

**Sec-mech** is an implementation of a security mechanism.

Figure 5.2 also shows another module that is affected by the introduction of security concern, that is the GSpace Proxy. A GSpace Proxy mediates the communication between an application component and a GSpace kernel. An application component and kernel could be located on different devices. It is the task of the proxy to hide from the application component the kernel location and all the details for establishing a communication channel with a kernel. Since the data that proxy and kernel exchange could be sensitive to the application, it is necessary to make such communication channel secure. Instead of modifying the code of the existing GSpace Proxy we placed a Sec-wrapper around the GSpace Proxy. This Sec-wrapper takes care of all the details necessary for establishing a secure connection with a kernel. Another important function of this Sec-wrapper will be presented in the next section, where the authentication phase is analyzed.
5.3 Architectural Design for Security

1 Tuple Type: AliceMsg
2 Authentication Mechanism: LocalUserPwdMech["Alice",Pwd];("Bob",Pwd)]
3 Authorization Mechanism: ACLMech["Alice",p,r,t];("Bob",p,r,−)
4 Storage Confidentiality: SCDesMech[key1]
5 Comm Confidentiality: CCDesMech[key2]

Figure 5.3 A sec-descriptor defined by Alice for tuple type AliceMsg.

5.3.3 An Example of a Security Policy

In this section, we provide an example of how a security policy is defined for a tuple type. We show how such a policy is enforced when a request for that tuple type is served. The policy is enforced by the activation of several security mechanisms. For each of those mechanisms we will describe all steps that are executed when each mechanism is activated.

Alice wants to exchange data with Bob via tuples of type AliceMsg. The tuple type should be accessible only by her and Bob. Thus she requires that each access to such a tuple type is authenticated and the only authorized users are herself and Bob. She wants to have full rights on the tuple type, meaning that she can execute put, read and take operations. She only allows Bob to execute put and read operations. Alice is also concerned with the confidentiality of her messages. She decides that tuples must be exchanged in encrypted form. Moreover, tuples must be stored encrypted.

The security policy is specified in a sec-descriptor. A somewhat simplified version of the actual implementation is shown in Figure 5.3. The first line in the example in Figure 5.3 defines which tuple type is affected by the security policy that the sec-descriptor specifies. The rest of the file specifies which mechanism must be activated for each security phase. Each mechanism is supplied with specific data. For instance, in line 2 it is specified that authentication is carried out using a login mechanism. Additionally, the table with user names and passwords is specified.

The sec-descriptor is downloaded in the kernels. Each kernel loads the file information into the local Sec-table.

Let us start describing what happens in the system when Bob executes a read operation. Figure 5.4 shows the execution flow when such a request is served. The setup consists of two kernels, A and B, running on different devices interconnected via a network.

Bob’s application requests kernel A to execute read operation. However, the request must be forwarded to kernel B because the tuple is not available on kernel A. This is executed transparently to the application. In the following, we analyze step by step how the execution flow propagates through the kernels’ modules.
Chapter 5 Security

Figure 5.4 Communication flow between different Sec-GSpace kernels.

- (1) A read with a template of type AliceMsg arrives at the Controller Sec-wrapper.
- (2) Before the request can be served, the requester must be authorized. The Controller Sec-wrapper notifies the Sec-manager of the new request and blocks the request until the Sec-manager replies.
- (3) The Sec-manager is aware that the requested read operation must be authenticated and the permissions checked. The Sec-manager looks in the Sec-table to find out which security mechanisms to activate for the authentication and authorization phases.
- (4) According to the policy that Alice specified, the user must be authenticated via a mechanism that needs user id and password. When the authentication mechanism is activated the user is required to supply his user id and password. Once the data has been supplied, the mechanism verifies it. Assuming that Bob supplied the correct (id, password) combination the mechanism acknowledges the Sec-manager that the requester has been authenticated. For the authorization phase, Alice specified that the Access Control List (ACL) mechanism should be activated. The Sec-manager supplies the type of operation, and the user id. Since it is Bob executing the read operation, the mechanism grants the access for the operation to be served. If either of the two mechanisms had not acknowledged the request the Sec-manager would have immediately informed the Controller Sec-wrapper to refuse the request. However, in this case, the Sec-manager informs the Controller Sec-wrapper to let the request proceed.
- (5) The Controller Sec-wrapper passes the request to the Controller to com-
5.3 Architectural Design for Security

The Controller forwards the request to the Dynamic Invocation Handler (DIH).

(6) - (7) The DIH selects the appropriate Distribution Policy. According to this distribution policy, the request has to be sent to another kernel. Therefore, it informs the Sec-wrapper of the Communication Module that a request must be sent to another kernels.

(8) The Communication Module Sec-wrapper notifies the Sec-manager. The Sec-manager has to activate a mechanism for confidentiality. The Sec-manager executes steps (3) and (4), this time activating the CDesMech mechanism for encrypting the request.

(9) Once the data is encrypted, the request is sent to kernel B. The Communication Module Sec-wrapper on kernel B has to decrypt the data before the Communication Module could serve it. However, if only ciphertext is sent, the Sec-wrapper will not be able to identify which mechanism should be used for decrypting the data. For this reason, the message contains an id that identifies the Sec-mechanism used for decrypting the data together with the ciphertext. Figure 5.5 shows the structure of such a message exchanged between kernels.

(10) The Sec-wrapper invokes the CDesMech mechanism identified via the id in the message to decrypt the request. Once the request is decrypted the Sec-wrapper passes the request to the Communication Module.

(11) The Communication Module forwards it to the Distribution Policy that has to retrieve the tuple locally.

(12) The Distribution Policy requests the Sec-wrapper of the Local Slice to retrieve a tuple.

(13) The Sec-wrapper notifies the Sec-manager on kernel B.

(14) - (15) The Sec-manager interprets the notification as a request to activate a mechanism for storage confidentiality. The Sec-manager looks in the Sec-table and activates the mechanism for decrypting tuples of type AliceMsg. Once the tuple is decrypted, the search in the local slice for a tuple matching the template is started. If the tuple is retrieved, it is sent back to the application following the inverse path.

Figure 5.5 Structure of a message exchanged between kernels.

<table>
<thead>
<tr>
<th>Sec-mechanism ID</th>
<th>Ciphertext</th>
</tr>
</thead>
</table>
Figure 5.6 Message sequence chart for the authentication phase.

Next phase

Save the context, log future requests

Verify data

Send back data

Forward CUI code

Send back user-id

Execute CUI

Forward CUI code

Send the CEG-

Request for operation

Notify the Sec-

Manage key

Type and operation

Secure the channel

Secure data

Type and operation

Check if the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-

Verify the context is already authenticated

Authentication method determined

Send the Sec-
In the following a detailed description of each mechanism activation is given.

**Authentication Mechanism Activation**

In this section, we concentrate on one type of security mechanism, that is the mechanism that deals with authentication. In particular, using the example above, we describe the steps that are executed when the authentication mechanism `LocalUserPwdMech` is activated. The `LocalUserPwdMech` is an authentication mechanism that uses user id and password for authenticating a user. Finally, we will list several alternatives of authentication mechanisms that could be used in our architecture.

Figure 5.6 shows the message sequence chart of the authentication phase. Bob’s application requests via a GSpace Proxy the execution of a read operation. The GSpace Proxy executed the operation on behalf of the application. The proxy forwards the request to the GSpace kernel to which it is bound. However, the sending of the request is mediated by the Proxy Sec-wrapper that is around the proxy. This Sec-wrapper has the task to establish a secure connection with Sec-wrapper of the Controller in the kernel. The connection established between the application and the kernel is identified by a unique context id. This is particularly useful when certain properties of the connection have to be asserted. The Controller Sec-wrapper notifies the Sec-manager about the request. The Sec-wrapper passes to the Sec-manager the context id, the tuple type and the type of operation. The Sec-manager looks up in the Sec-table and finds out that requests for tuple type `AliceMsg` must be authenticated. The Sec-manager checks whether the requester was already authenticated by querying the Context-table with the given context id.

The Context-table contains the context id and the status (such as if it has been authenticated and authorized) for each active connection between the kernel and the application components. Each kernel has its own Context-table. When a new connection is established a new context id is generated and assigned to the connection. A context id is valid within the context of a connection between the application that made the request and the kernel that is serving it. An application component can have only one active connection open at a time with a kernel. When an application component closes the connection with a kernel the respective context id is removed from the Context-table of that kernel. All details of generation, assignment and disposition of a context id are hidden from the application component.

Since this is the first request that comes from Bob’s application, the status of the connection stored in the Context-table states that this application has not been authenticated yet. Thus, the Sec-manager has to activate the mechanism for authentication. The mechanism that is associated with tuple type `AliceMsg` is

\[1\] We assume that the specific cryptographic mechanism running on kernel B has some means for retrieving the correct key for decrypting the data.
**LocalUserPwdMech.** This mechanism requires a user id and password that is verified by looking it up in the local User-Pwd-table.

The gathering of data necessary to the mechanism for the authentication is not trivial. For instance the **LocalUserPwdMech** uses a GUI that asks the user id and password. The GUI has to be shown on the device that the user is using for running the application. Since the code for retrieving this information is related to the specific mechanism, the security code should not be entangled with the application code. This has the following advantages:

- First of all, the writing of application code is easier since the application developer has only to concentrate on the functional concern of the application.

- Secondly, it increases the reusability of application code, for the application can be deployed in different environments with different security mechanisms for authentication.

- Finally, it increases the flexibility of our architecture, because it can be deployed with applications that have different security needs or it can support changes of the needs of a given application.

The security mechanism should be responsible for providing the **collector code** to the user/application device. However, once the collector code reaches the target device it needs to be activated. And it is here that the Proxy Sec-wrapper comes to help. The Proxy Sec-wrapper takes care of executing the collector code and eventually sending back the necessary information to the mechanism. We assume that the collector code provided by a security mechanism is trusted.

There are two possible ways in which the collector code can be made available:

1. The collector code could be already available inside the Sec-wrapper of the GSpace Proxy. The Sec-wrapper has to activate a specific collector code when requested by the kernel.

2. Alternatively, the collector code is in the kernel within the mechanism that requires it. When the collector code must be activated, the mechanism forwards the code to the Sec-wrapper. The Sec-wrapper just has to activate it.

The first option has the advantage that the code is already available where it is needed. However, for all security mechanisms that require a specific tool (such as a GUI) the related collector code should already be available in the Sec-wrapper. The second option is more flexible, in the sense that when a new mechanism is downloaded in the kernel all the required code is downloaded as well and made available when necessary\(^2\). On the other hand, sending the specific code to the

---

\(^2\)A combination of the two options is also possible. The code could reside on the Security
application location increases the latency and requires bandwidth. In the following, we assume that the second option is adopted.

The Sec-manager activates the `LocalUserPwdMech`. The `LocalUserPwdMech` sends its collector code to the Sec-wrapper. When the collector code arrives to the proxy Sec-wrapper it is executed and the GUI is displayed to the user. After the user inserts the data, the data is sent back to kernel. The authentication mechanism verifies the information in its local data structure. If the data is correct, the mechanism informs the Sec-manager that the request can proceed to the next phase.

The next phase consists in checking whether the user has the rights to execute the requested operation. Details of the authorization phase are discussed later in this section. In the following, we list several other authentication mechanisms that could be supported by our framework.

**User ID and Password with External Verification** Another authentication mechanism could be the following. An Authentication Authority (AA) is an application that is responsible of storing and verifying user id and password. The authentication mechanism could use the services of the AA for the verification phase. After the user information is collected, the authentication mechanism establishes a connection with the AA and the data is sent over for verification. Assuming that the data is correct, the AA sends back an acknowledge message to the authentication mechanism. Thus, the operation can proceed as for the previous case. However, the authentication mechanism needs a communication means for contacting the AA. These communication means can be found already in the Sec-wrapper around the Controller. In fact, this Sec-wrapper is already responsible for all secure communications between kernel and applications. Since the AA is just an application, the Controller Sec-wrapper is the most suitable module for handling this type of communication.

**Security Key** This authentication mechanism uses a secret key to authenticate a user. A user is supplied with a SmartCard where a secret key is stored. When this mechanism is activated, the collector code reads the key from the user’s SmartCard and verifies which user is associated with the key. Since the key is verified within the architectural boundaries of a kernel we can guarantee that the key will not be misused. This mechanism requires that a SmartCard reader is available on the device where the key is collected.

**Authentication with Certificate** This mechanism uses certificates for authenticating entities. A certificate can state the role or identity of its owner entity. For
Biometrical Authentication  For authenticating a user her physical unique features could be used, such as fingerprints, iris image, face characteristics and even DNA. This type of authentication requires specialized hardware (for instance, scanners for fingerprint and iris) to capture the salient features of the user. If we assume that such hardware is available then we can devise a security mechanism in GSpace that uses biometrical data for authenticating a user. When a user requests an operation, the authentication mechanism sends the specific collector code. For instance, the collector code could consists of a GUI to ask the user to perform biometrical measurements. If fingerprints are used, the GUI could ask the user to put his finger(s) on the scanner. Once the data is captured, the data is sent back for verification.

Combination of Different Authentication Mechanisms  Sometimes it might be useful to be able to use several types of authentication mechanisms for the same tuple type. Thus, the application developer could decide that for a higher degree of security Biometrical Authentication should be used. However, not all devices have the proper hardware necessary for collecting biometrical data from a user. In that case, the mechanism that uses user id and password could do. Therefore, a composed authentication mechanism can be devised: when activated check the available hardware on the user device. If dedicated hardware for biometrics is available then user biometrical authentication mechanism. Otherwise, adopt the user id and password mechanism.

Authorization Phase in Details

In this section, we discuss in more details the steps that are executed for the authorization phase. We first concentrate on the ACLMech, that is the authorization mechanism used by Alice. Afterwards, we show another authorization mechanism that could be used.

Figure 5.7 shows the message sequence chart of the authorization phase. The Sec-manager looks in the Sec-table for retrieving which authorization mechanism should be used for the tuple type AliceMsg. The authorization mechanism associated with the tuple type is ACLMech. The ACLMech is an authorization mechanism based on an access control list which maintains for each user id a list of access rights. The Sec-manager activates the ACLMech and passes it the following parameters: the
5.3 Architectural Design for Security

Figure 5.7 Message sequence chart for the authorization phase.

tuple type, the user id, and the operation that should be executed. The ACLMech retrieves the access rights that are associated with the given user id and tuple type from the Access Right Table (ART). Since the tuple type is AliceMsg, the user is Bob, and the operation is a read the operation can be executed. Thus ACLMech acknowledges the Sec-manager to proceed with the operation.

Capability Authorization Mechanism Another mechanism for authorization could use capabilities. A capability is an unforgeable identifier that is associated with a resource for which it specifies the access rights of the capability holder. The first time a request comes from a user for a tuple type the capability must be generated. The CapMech generates a random number, check, that is uniquely associated with the user id and tuple type. The check is XORed with the access rights associated with the user contained in the ART. The value obtained from the XOR is run through a one-way hash function. This value is stored in the check field of the capability. Another field of the capability is for storing the access rights obtained from the ART. Figure 5.8 shows how the capability for user Bob and tuple type AliceMsg is generated. Once the capability is generated it is given back to the Sec-wrapper around the proxy of the application. When other requests for operations are generated (for the same tuple type and same user), the Security Wrapper passes along with the request the capability that was previously obtained. When the capability is passed to the CapMech, it need only check whether the value in the check field is correct. In that case, the access rights that are in the capability can be used for granting the request. This avoids that for every request
Confidentiality Phase in Details

The last phase makes sure that the tuples are not usable even if they get intercepted or stolen. This phase takes place when tuples are either sent over to another kernel or stored in the local slice.

Confidentiality for Communication When tuples (or templates) are exchanged between kernels, they could be transmitted in ciphertext to avoid eavesdropping attacks. The Communication Module is responsible for inter-kernel communications. However, the Communication Module is not equipped to deal with encrypted communications. Thus, when a request must be sent to another kernel the Sec-wrapper around the Communication Module informs the Sec-manager for finding out whether the request for tuple type AliceMsg should be encrypted. The Sec-manager retrieves this information form the Sec-table. As specified by Alice, communication between kernels must be encrypted using mechanism CCDesMech, that is a mechanism that uses the DES algorithm for encrypting data. The data for the request is encrypted and sent to kernel B. Together with the data, the id of the security mechanism that was used, are sent to kernel B as showed in Figure 5.5. When the message is received in kernel B by the Sec-wrapper of the Communication Module, the Sec-wrapper uses the id part of the message to find out which mechanism for the decryption should be used. In this case, the CCDesMech is used for decrypting the data of the request. Once the plaintext is obtained, the request is passed to the distribution policy that takes care of performing the read on the local slice.

Figure 5.8 The generation of a capability.
Confidentiality for Storage  Since the data stored in the slice could be sensitive to its owner, confidentiality must be taken into account. A way for protecting the data is the use of encryption techniques. Again, we did not modify the implementation of our current slice. We use a Sec-wrapper to handle the security details. When a tuple is inserted in the local slice in a kernel, Sec-wrapper notifies the Sec-manager for knowing whether the tuple must be stored in plaintext or encrypted. For tuples of type AliceMsg the SCDesMech should be activated. The search starts by selecting all tuples of type AliceMsg contained in the slice. Afterwards, by using the mechanism’s decryption method tuples are decrypted and matched with the given template. Eventually, the matching tuple is found and returned to the user.

We distinguish two types of storage confidentiality: volatile and permanent. Confidentiality for volatile storage refers to the case when tuples are stored in volatile memory. These exist as long as either the kernel is running or the host device is on. Otherwise the tuple content of a kernel is lost. Clearly in this case, the confidentiality mechanism must encrypt and decrypt a tuple each time an access is executed. The second type of confidentiality refers to the case when the tuple content of a kernel must be kept even if a kernel is not running. In this case, the mechanism is activated each time a kernel is started or stopped. When a kernel is started, the tuple content is loaded from an encrypted file to the local slice of the kernel and vice versa when the kernel is stopped.

5.4 Managing Security Properties

The introduction of security concerns in the SDS model requires that either the users or the administrators take care of some security management tasks. What is important to notice is that these tasks are completely separated from the application logic. First of all, it is required that a user creates the space and de-
fines security policies for its tuple types. We call this user space owner. After
the space is instantiated, then the space owner must configure the space. This
means choosing the kernels/devices to be assigned to the space, specifying which
users/applications are authorized to access the space, and assigning the rights to
each of them. Finally, the space owner must decide whether or not to enforce
confidentiality.

Another important aspect of managing security settings is restricting access to
those very settings to the space owner.

As we saw in the previous section, Alice is required to write a security descriptor
file and download it to each kernel that she wants to assign to her space. What
we saw was a somewhat simplified version of the real descriptor. Not only is such
a descriptor more complicated to write but it is also tedious and error prone.
Moreover, any modification requires the changing of the descriptors in all kernels.

To reduce the effort for (re-)configuring the system, we developed a tool that
provides space owners with a GUI that helps in the configuration of a space. We
named this application Security Manager Console (SMC). Figure 5.9 shows
the main panel of the SMC. The SMC provides users with several functionalities,
such as:

- Scanning the environment for finding GSpace Kernels
- Retrieving from a kernel its actual security information
- Creating a new secure space
- Modifying the security settings of an exiting space

The text area on the left-hand side provides the spaces that are assigned to a
given kernel. The kernels found in the proximities are listed in the text area on
the right-hand side. Clicking one of the kernels in this list triggers the listing of
spaces assigned to that kernel.

In the next section, we present a case study that allowed us to validated the design
decision of our security framework.

5.5 Personal Health Systems—The Philips Case
Study

Part of the research was carried out at Philips Research Eindhoven, The Nether-
lands. One of the research topics currently conducted at Philips is that of Personal
Health Care Management (PHCM). PHCM focuses on remote assistance for pa-
tients with minor diseases from specialized health care centers, such as hospitals.
PHCM can be divided in two main subjects:
• Supplying the patient’s daily environment with the appropriate equipment for collecting and monitoring data on her health status

• Providing a back-end service where the data from different patients could be collected and analyzed by health care specialists. Moreover, remote feedback could be provided to the patient from the health centers without having the patient leaving her environment.

We focus on the first subject, with particular emphasis on the security of the patient’s data when it is distributed across the patient’s devices.

This section describes how by means of GSpace the data could be distributed across such devices and protected according to security policies that the patient determines. In the following, we discuss the motivations and assumptions that have driven the development of a framework for PHCM based on GSpace. We continue with the description of some user scenarios and the corresponding applications that we developed. We conclude with a description of the configuration of the system and its deployment.

5.5.1 Motivations and Assumptions

Demographic developments in society lead to an increasing demand for health care. The main reasons are that people get older and become increasingly aware of disease risks and importance of a healthy lifestyle. At the same time health care budgets are under pressure. These two developments demand new ways of delivering health care services.

One approach is to support people in having a healthy life or managing their disease while residing outside traditional health care institutes, i.e. at home. In this case, sensor-based systems may continuously acquire data on a person’s status and context. These data can be interpreted locally by the system, which may give immediate feedback or control an actuator, or be sent via a network to a remote care provider for further interpretation.

A system that supports an individual’s health is called a personal health system (PHS). It consists of sensors, actuators, and appliances (such as cellular phones and PDAs) within the range of an individual person. A PHS may be connected to other systems, such as hospital information systems, tele-healthcare services, or fitness centers.

A PHS typically has multiple applications that run on different devices but that need to share data on a person’s status and context. For instance, weight measurements could be shared by a weight-management application running on the users PC at home and a diabetes application running on the users cell phone. Hence these data need to be distributed to the correct devices. However, distribution of
data cannot be predicted a-priori since usage-patterns can change and new devices, applications, and sensors can be added to a PHS.

Currently, data distribution in PHSes is done in a point-to-point manner and often in such a way that applications are aware of how the data should be distributed. The SDS model is well suited for this type of application. With its decoupling in time and space, building and maintaining such an application becomes easier. As a matter of fact, the model abstract from issue such as data distribution that becomes transparent to the application components.

However, the basic SDS model does not address security. In PHSes security is crucial. Security not only guarantees the dependability of the PHS application but more importantly it could protect the life of patients. As a matter of fact, a patient’s space could contain medication prescriptions from the patient’s GP. Without security, a malicious user could threat the patient’s life by inserting in the patient’s SDS a fake prescription for a hazardous medication.

With the security extensions presented in this chapter, GSpace can be deployed in a PHS. Moreover, the practicing SoC for security and distribution concerns eases the building of applications for a PHS.

In the following, we list several assumptions that we made concerning the environment where the PHS is deployed:

1. All devices taken into consideration in the case study should be considered as GSpace-compliant, meaning that a well-behaving GSpace kernel is running on each device.

2. There is a back-up service that takes care of making back-ups of the data in a secure manner. In case of data loss, the content of the SDS can be restored to the latest saved back-up.

3. The GSpace kernel can enforce security as long as tuples do not leave its boundary of control. Once tuples are given to applications or to users then there is nothing that a kernel can do to prevent that tuples are misused.

4. For simplicity, we assume that an application and the user using it can be considered as the same entity. In this case, we can group entities as follows:

   - Trusted entity: the owner of the space is willing to communicate information with this entity.
   - Untrusted entity: the patient does not trust this entity and therefore no information should be provided.

For our case, we only deal with trusted entities. Untrusted entities simply do not get any access to the content.
5. With respect to a given space $s$, GSpace kernels can be grouped in two categories:

- **Internal Kernel**: kernels that are assigned to $s$. When a kernel is assigned to a space the kernel must reserve resources for this space (such as storage for its tuples and security settings). These kernels collaborate with each other for enforcing the desired security policies over $s$.
- **External Kernel**: kernels that are not assigned to $s$.

We assume that kernel-to-kernel requests for tuples belonging to $s$ can only be issued by internal kernels.

6. Devices where a kernel could be deployed can be divided in two classes:

- **Trusted Device**: a device in which the data can be stored without being under the threat of losing its confidentiality.
- **Untrusted Device**: a device from which the data can be retrieved without user permission. This could be either because the device is administered by a malicious entity or because the device can be easily lost or stolen.

Tuples stored in kernels running on untrusted devices are always stored encrypted.

7. Communication links between nodes can be categorized as follows:

- **Secure link**: a link that is protected against eavesdrop attacks.
- **Unsecure link**: a link that does not provide any protection against eavesdrop attacks.

We assume that communication links are by default not secure. As default, the communication channels between application components and kernels are established via the standard Secure Socket Layer.

### 5.5.2 User Scenarios

In this section, we describe the user scenarios that we used for building a demo for a PHS based on GSpace.

Cindy and Jim are a couple that live in the same household and that want to take care of their health with the support of a PHS. Since they live in the same household some of the devices where the PHS is running could be shared. In the following, we present scenarios for each of them.
Cindy’s Scenario  Cindy wants to lose some weight. Every morning she stands on the weight scale at home for collecting her weight. The scale that Cindy uses at home is the latest model with some computation capabilities. For instance, this smartscale is able to recognize the user via a card reader. Cindy has a SmartCard that she uses for identifying herself when she steps on the scale. Although the data is not very sensitive, she doesn’t want Jim to see this graph.

When Cindy goes on a business trip and stays in a hotel she can only use the scale that is in her hotel room. However, the scale in the room is just a normal scale. The weight values that Cindy collects when she is away from home are stored in her mobile phone. When she returns home, the data of her mobile is made available to the other devices.

Cindy also wants that her dietician, Dr. Monica, gets access to her weight values.

Jim’s Scenario  Several times a week Jim feels dizzy. His general practitioner, Dr. Rachel, suspects a small cardio-vascular problem, and wants Jim to collect for a period his heart rate, blood pressure, and weight values. The sensors for heart rate and blood pressure have a limited computation power but they are able to transmit data over a wireless connection. In this way, the data can be stored on Jim’s mobile phone.

When Jim is in the shopping mall, he feels very dizzy and calls his GP. Dr. Rachel uses her PC to remotely access Jim’s data from the medihub that is at his home. She can check his heart rate, blood pressure, and weight. She sees no abnormalities, but advises him to sit down for a while.

After Jim has sat down for a while, he continues shopping. As he feels very dizzy again, the pet shop owner calls an ambulance. While the ambulance is driving to the shopping mall, the paramedic uses the on-board computer of the ambulance to remotely access Jim’s data, and inspects his heart rate and blood pressure. Jim is brought to the hospital, but can go home the same day.

Jim is concerned with the confidentiality of his data. He wants to grant access only to his GP. However, in case of an emergency, he wants that hospital personnel (included paramedics in the ambulance) can get access too.

5.5.3 Deployment of a PHS Based on GSpace

In this section, we describe how Cindy and Jim deploy and configure their PHSes. As a first step, GSpace kernels should be installed on the patients’ devices. Each GSpace kernel is designed such that it can support multiple separate spaces. Thus, even if a device is shared by several spaces only one kernel is necessary. Once the kernels are installed, Cindy and Jim can configure the security policies of their spaces using the SMC described in Section 5.4. We assume that all devices in the scenario have some means for communications.
5.5 Personal Health Systems–The Philips Case Study

<table>
<thead>
<tr>
<th>Device</th>
<th>GSpace Kernel</th>
<th>Application Component</th>
<th>Shared</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mobile Phone</td>
<td>Yes</td>
<td>WeightSampler</td>
<td>No</td>
</tr>
<tr>
<td>Smartscale</td>
<td>Yes</td>
<td>SWeightSampler</td>
<td>Yes</td>
</tr>
<tr>
<td>Medihub</td>
<td>Yes</td>
<td>n/a</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 5.2 Properties for the devices used in Cindy’s scenario.

<table>
<thead>
<tr>
<th>User/Application</th>
<th>WeightTuple</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cindy</td>
<td>p,r,t</td>
</tr>
<tr>
<td>Dr. Monica</td>
<td>-,r,-</td>
</tr>
</tbody>
</table>

Table 5.3 Access rights for users in Cindy’s scenario.

Configuring Cindy’s Space

The devices that Cindy uses for her PHS are listed in Table 5.2. All the devices have enough computation capabilities for running a GSpace kernel together with an application component.

Cindy runs the SMC on her laptop; the laptop is not part of Cindy’s PHS. Cindy requests the SMC to create a new SDS. This brings up a dialog for a new SDS. Cindy inserts the name of her SDS, and the user name and password for the space manager. Afterwards, Cindy selects the kernels to be assigned to her space. Using its discovery mechanism, the SMC provides a list of active GSpace kernels running on the connected devices at home. Cindy selects the kernels running on the following devices: her mobile phone, the smartscale and the medihub.

Cindy defines the tuple type WeightTuple for storing the weight samples in her PHS. She defines for WeightTuple the following security policy:

- only authorized users can access the tuple type
- no loss of confidentiality should happen during data communication
- no loss of confidentiality should happen if her mobile phone is lost

She sets as authorized users herself with full rights and her dietitian, and Dr. Monica, with only read rights. Table 5.3 summarizes the access rights in Cindy’s scenario.

Several mechanisms for authentication are available. Some of these mechanisms have specific requirements about the hardware necessary for executing the authentication procedure. For instance, a biometrical mechanism based on retina map might required a retina scanner. When Cindy selects the authentication mechanism for her space, she has to match the requirements of the mechanisms with the hardware specifications of the devices in her system. Giving the set of devices that Cindy assigned to her PHS the following hardware specifications are available:
<table>
<thead>
<tr>
<th>Device</th>
<th>GSpace Kernel</th>
<th>Application Component</th>
<th>Shared</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blood Pressure Sensor</td>
<td>No</td>
<td>BPSampler</td>
<td>No</td>
</tr>
<tr>
<td>Heart Rate Sensor</td>
<td>No</td>
<td>HRSampler</td>
<td>No</td>
</tr>
<tr>
<td>Mobile Phone</td>
<td>Yes</td>
<td>WeightSampler</td>
<td>No</td>
</tr>
<tr>
<td>Smartscale</td>
<td>Yes</td>
<td>SWeightSampler</td>
<td>Yes</td>
</tr>
<tr>
<td>Medihub</td>
<td>Yes</td>
<td>na</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Table 5.4** Properties for the devices used in Jim’s scenario.

- Keyboard and Display for her mobile phone
- Security Card Reader for the Smartscale and Dr. Monica’s PC
- not available for the medihub

The SMC provides a list of authorization mechanisms that are compatible with the devices’ specifications. Cindy selects from this list the following mechanisms: `LocalUserPwdMech` and `SecCardMech`.

Cindy selects the `CCDesMech` for ensuring that no loss of confidentiality might happen during data communication. She sets her mobile as untrusted, forcing the kernel running on the mobile phone to store tuples in the local slice encrypted. In this way, even if her mobile is either lost or stolen, confidentiality is ensured.

**Configuring Jim’s Space**

The devices that Jim uses in his PHS are listed in Table 5.4. The Blood Pressure Sensor and the Heart Rate Sensor do not have enough capabilities for running a GSpace kernel. Thus, the application components running on these devices have to store their tuples into a remote GSpace kernel via the GSpace Proxy.

Jim starts the SMC on the home PC; the home PC is not part of his PHS. Jim creates a new space using the SMC. Jim sets the space name and owner properties. Afterwards, Jim selects the kernels for his space. Jim selects the kernels running on the smartscale, on his mobile phone and on the medihub.

Jim has to monitor his cardio-vascular parameters such as heart rate and blood pressure. Moreover, some information on his weight is also required. He defines the following tuple types: `HRTuple` for heart rate samples, `BPTuple` for blood pressure samples, and `WeightTuple` for weight samples. He defines for these tuple types the following security policy:

- only authorized users can access all tuple types
- no loss of confidentiality should happen during communication for tuple types `HRTuple` and `BPTuple`. 
5.5 Personal Health Systems–The Philips Case Study

- no loss of confidentiality should happen if his mobile phone is lost for tuple type HRTuple and BPTuple.

He sets as authorized user for all tuple types himself with full rights. For his GP, Dr. Rachel, Jim sets only read access to all tuple types. Also, he grants read access to all tuple types to the hospital personnel, in case he is taken into a hospital for an emergency. Finally, to his dietitian, Dr. Ross, he grants read access only for tuple type WeightTuple. Table 5.5 summarizes the access rights specification for the users and applications in Jim’s scenario.

Jim must select the authentication mechanism to be used to authenticate users. Jim sets for all tuple types the LocalUserPwdMech. However, such a mechanism is only feasible for the GP and dietician cases. The reason is that Jim knows his GP and his dietician thus he can set a user id and password and provides them to the respective users. This is not well suited for the hospital personnel case, since it would require that Jim should give a password for all hospital personnel working for the hospitals in the area where he lives.

The CertMech is well suited for this type of access. Jim sets the digital signature of the authority that signs the certificates. In this case it would be the Hospital Authority. Afterwards, he selects to which roles he wants to grant access. The roles are given by the authentication authority. For instance, for a hospital ER we can have: er-doc, er-nurse and paramedic. Jim selects the CertMech in the security mechanism dialog of he SMC. This action brings up the dialog specific for this type of mechanism and that is shown in Figure snapshot6. Jim sets the role of the entity to which he wants to grant access. Afterwards, he has to point to the public key of the authority that has to sign the certificate.

Another situation where the CertMech is well suited is the following. Jim wants to authenticate the application components running on the blood pressure and heart rate sensors. These devices do not support any interface, except for a button to turn them on and off. It would be convenient that the authentication is carried out without user intervention. We can assume that these devices can have a unique id and an embedded certificate that was signed by the Hospital Authority. During the configuration of his space, Jim has to insert the sensors’ id in the dialog of the CertMech. Moreover, he has to select the public key of the Hospital Authority that signed the certificates embedded in the sensors.

As a last step, Jim sets the confidentiality settings. Jim selects for tuple types BPTuple and HRTuple the DESStorageMech for storage encryption and the CCDesMech for communication encryption. No confidentiality is applied to WeightTuple, since for Jim the weight value is not such a sensitive data.
<table>
<thead>
<tr>
<th>User/Application</th>
<th>WeightTuple</th>
<th>HRTuple</th>
<th>BPTuple</th>
</tr>
</thead>
<tbody>
<tr>
<td>BPSampler</td>
<td>–</td>
<td>–</td>
<td>p,-,-</td>
</tr>
<tr>
<td>HRSampler</td>
<td>–</td>
<td>p,-,-</td>
<td>–</td>
</tr>
<tr>
<td>Jim</td>
<td>p,r,t</td>
<td>p,r,t</td>
<td>p,r,t</td>
</tr>
<tr>
<td>Dr. Rachel</td>
<td>-,r,-</td>
<td>-,r,-</td>
<td>-,r,-</td>
</tr>
<tr>
<td>Dr. Ross</td>
<td>-,r,-</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>paramedic</td>
<td>-,r,-</td>
<td>-,r,-</td>
<td>-,r,-</td>
</tr>
<tr>
<td>er-doc</td>
<td>-,r,-</td>
<td>-,r,-</td>
<td>-,r,-</td>
</tr>
<tr>
<td>er-nurse</td>
<td>-,r,-</td>
<td>-,r,-</td>
<td>-,r,-</td>
</tr>
</tbody>
</table>

Table 5.5 Access rights for users and applications in Jim’s scenario.

5.5.4 Collecting Data from Devices

In this section, we describe how application components exchange tuples with a space. To prove the power and versatility of our system, we will present several scenarios giving details of how security policies are enforced.

Collecting Weight Samples for Cindy

This section focuses on the weight management scenario for Cindy. We first describe the case when Cindy is at home using the smartscale. Afterwards, we will focus on the case when Cindy is away from home and has to use a normal scale.

When Cindy is at home she uses the smartscale for collecting her weight samples. The smartscale has a SecCard reader that is used for identifying the user via her secret key stored in a SecCard. The smartscale has some computational power and a GSpace kernel is running on it. The kernel was assigned to Cindy’s space.

The following describes what happens when Cindy steps on the scale:

1. The application component SWeightSampler asks Cindy to insert her SecCard. Once the application finds out that it is Cindy, it collects the value that the scale produces. Note that the authentication phase of the SWeightSampler is not part of the security settings of Cindy’s space. The application component instantiates a tuple of type WeightTuple with the weight value just collected. The application component executes a put operation with this tuple.

2. The ProxyWrapper intercepts the operation. Before forwarding the operation to a kernel, the ProxyWrapper has to find out to which kernel the operation should be forwarded. Moreover, the ProxyWrapper must specify which space among those instantiated in that kernel is the recipient of the operation. All these details are hidden from applications by the ProxyWrap-
The ProxyWrapper is designed in such a way that it is able to support several strategies that we will present during this discussion. The user has to set which strategy to use in a file. For this specific case, Cindy sets as default kernel the local kernel. In this way, the ProxyWrapper forwards all the operations coming from the SWeightSampler to this kernel. However, this kernel is assigned to manage two spaces. To find out which of the spaces should be selected Cindy specified an option in the property file. This option tells the ProxyWrapper to ask the user for indications. The ProxyWrapper displays on the HUD of the scale a simple GUI with the available choices. Cindy, using the four-way button selects the option that represents her space and presses ok. At this point, the ProxyWrapper can forward the request to the local kernel and for Cindy’s space. The ProxyWrapper also sends together with the request the hardware specification for its device. In this specific case, the specification will tell the kernel that the device has a SecCard reader.

3. The request is intercepted by the ControllerWrapper. The ControllerWrapper notifies the SecManager that the following request arrived: put operation with tuple type WeightTuple to be executed on CindySpace. The SecManager looks up in the SecTable of CindySpace for the authentication mechanism(s) that should be activated. The SecManager finds out that the following mechanisms are associated with type WeightTuple:

   - LocalUserPwdMech; requires keyboard and display
   - SecCardMech; requires SecCard reader

   Since the scale has a SecCard reader the SecManager selects the SecCardMech for activation.

4. On activation, the SecCardMech forwards to the ProxyWrapper the code for collecting the user’s key from the SecCard. When the code arrives, the ProxyWrapper activates it. Afterwards, the key is retrieved and sent to the SecCardMech for verification.

5. Assuming that Cindy provided her SecCard, the put operation is executed and the tuple is stored locally on the scale’s kernel. Afterwards, the tuple is distributed to other kernels according to the distribution policy.

When Cindy is in a hotel room she does not have access to her smartscale. The hotel room has a regular scale. For storing the weight into her space, Cindy uses her mobile phone. The mobile has a GSpace kernel running together with the WeightSampler. Cindy uses the scale of the room for sampling her weight. She inserts manually the value that the scale read to the WeightSampler, using the phone keypad. After she confirms the data, the following steps are executed:
1. The WeightSampler instantiates a tuple of type WeightTuple with the value inserted by Cindy and it executes a put operation.

2. The operation is intercepted by the ProxyWrapper. The ProxyWrapper has to determine the kernel and space where the operation should be sent. Cindy, in this case, set as default kernel the local kernel and as default space her space (since no other spaces are available on this kernel). The ProxyWrapper reads the default values from a file and sends the request directly without asking for user intervention. The wrapper also sends the hardware specification for the mobile phone, specifying that a display and a keyboard are available.

3. The request is intercepted by the ControllerWrapper. The ControllerWrapper notifies the SecManager that the following request arrived: put operation with tuple type WeightTuple to be executed on CindySpace. The SecManager looks in the SecTable of CindySpace which authentication mechanism(s) should be activated, finding:
   - LocalUserPwdMech; requires keyboard and display
   - SecCardMech; requires SecCard reader

   In this case the SecManager selects the LocalUserPwdMech for activation.

4. The LocalUserPwdMech forwards the GUI code to the ProxyWrapper. When executed by the ProxyWrapper, the GUI is displayed on the mobile phone’s display. Cindy inserts her user name and password using the phone’s keyboard. This information is sent to the kernel where the LocalUserPwdMech locally verifies the authenticity.

5. The tuple is stored in the local slice and it is encrypted (as part of the policy that Cindy specified). The distribution policy is then applied. However, there are no other devices in range at this time. The distribution policy will take care of synchronizing the content.

Collecting Blood Pressure Samples for Jim

In this section we discuss how the sensors used in Jim’s scenario insert the data into Jim’s space. Here we focus only on the blood pressure sensor since the same description is valid for the heart-rate sensor.

When Jim wants to collect samples of his blood pressure, he activates the sensor. This automatically starts the application component BPSampler and the following steps are executed:
1. When the data of the blood pressure is available, the BPSampler instantiates a tuple of type BPTuple with the collected values. The BPSampler executes a put operation.

2. The operation is intercepted by the ProxyWrapper. Jim set the property file of the wrapper in such a way that the operation is forwarded to the kernel running on his mobile (since the sensor does not have a kernel) and to his space. The wrapper also requires that the CertMech should be used.

3. The request is intercepted by the ControllerWrapper. The ControllerWrapper notifies the SecManager that the following request arrived: put operation with tuple type BPTuple to be executed on Jim’s Space. The SecManager looks up the SecTable of which authentication mechanism(s) that should be activated. The SecManager finds out that the following mechanisms are associated with type BPTuple:
   - LocalUserPwdMech; requires keyboard and display
   - CertMech; no requirements

   Since the request specifies that the CertMech should be used, the SecManager selects the requested mechanism.

4. The mechanism retrieves the certificate from the sensor and verifies its authenticity. Assuming that an authentic certificate was provided, the tuple is stored in the phone kernel. Afterwards, the distribution policy takes care of replicating the tuple over other devices in range.
Chapter 5 Security

Figure 5.11  Snapshot of ConnectionManager GUI.

Accessing Patient Space

In this section, we describe how health specialists retrieve health data from the space of patients. More precisely, we present the case of Dr. Rachel, Jim’s GP, getting access to Jim’s blood pressure values.

For allowing health specialists to access patient’s data we developed the GPMedicalBrowser application component. Figure 5.10 provides a snapshot of an instance of the GPMedicalBrowser running on Dr. Rachel’s PC. The application can remotely access a space instantiated in a kernel running on one of the patient’s devices via the ProxyWrapper.

In the following, we list the steps that are executed for accessing the data from Jim’s medihub:

1. Dr. Rachel starts the GPMedicalBrowser on her PC in her office. Dr. Rachel clicks on the button on the GPMedicalBrowser’s GUI for retrieving a blood pressure values. The GPMedicalBrowser executes a read operation with a template of type BPTuple.

2. The ProxyWrapper intercepts the request. The ProxyWrapper has to find out where the request should be sent. For allowing Dr. Rachel to manage the connection with several patient’s spaces this wrapper was programmed to pop up a more complex GUI than the one used for the smart-scale case. We called this GUI ConnectionManager. Figure 5.11 provides a snapshot of the ConnectionManager. This application is independent from the GPMedicalBrowser. Dr. Rachel selects the patient’s space and the request is forwarded to Jim’s medihub. In the current implementation it is necessary to insert the address of the medihub at Jim’s house. The medihub is similar to a home gateway thus it is assumed to be always up. The ProxyWrapper also
communicates that the PC has keyboard and display.

3. When the request arrives at the ControllerWrapper of Jim’s medihub, this notifies the SecManager that a read operation on tuple type BPTuple arrived. The SecManager finds out that the following mechanisms could be activated for authentication for the given tuple type:

- LocalUserPwdMech; requires keyboard and display
- CertMech; no requirements

According to the request of the WrapperProxy, the SecManager activates the LocalUserPwdMech.

4. Once the LocalUserPwdMech is activated, it sends to the ProxyWrapper the GUI for collecting the user name and password. Dr. Rachel inserts the information required and can be authenticated.

5. After the authentication phase, the SecManager activates the authorization mechanism to check if the user has the required access rights. The mechanism grants access since Jim specified that user Dr. Rachel has read access for tuple type BPTuple.

6. The read operation is executed locally. The LocalSliceWrapper notifies the SecManager that an operation must be executed. The SecManager finds out that the tuples of type BPTuple are stored encrypted. The SecManager retrieves the decryption mechanism from the SecTable. The SDesMech is selected and used for decrypting the tuple. The tuple in cleartext is returned to the requesting application.

Let us consider the case when Jim requires the intervention of an ambulance. In this case, the identity of the user that wants to access Jim’s space is not known to Jim. The following steps are executed by a paramedic that wants to access Jim’s SDS from an ambulance.

1. On the ambulance’s terminal the paramedic executes the GPMedicalBrowser for accessing patient data. The application executes the read operation for tuple type BPTuple.

2. The ProxyWrapper intercepts the operation. The paramedic specifies which space to connect to using the connection manager GUI. The identity of the patient (together with the space name and address of Jim’s medihub) is downloaded to the ambulance’s computer when the request to intervene is accepted by the hospital’s emergency call center. The wrapper also requires that a certificate mechanism should be used.
3. When the request arrives at the kernel running on Jim’s medical hub the ControllerWrapper notifies the SecManager. The request is for a read operation on tuple type \texttt{BPTuple}.

4. The SecManager retrieves which authentication mechanisms are associated with the given tuple type it activates the \texttt{CertMech}.

5. Once activated, the mechanism forwards the code for collecting a certificate that states that the user accessing the space is a \textit{Paramedic}. The \texttt{CertMech} verifies that the signature of the certificate is valid.

6. Once the role is authenticated, the SecManager checks the access rights. Since Jim specified that paramedics have read access for tuple type \texttt{BPTuple} the request can be accepted.

7. Before sending the tuple to the paramedic application, the tuple is decrypted as in the previous case.

5.6 Related Work

In this section a collection of works on SDS with security mechanism is presented.

\textbf{Security Extension in Lime} In the work described in \cite{33}, Lime \cite{59} is extended with security and privacy. Since Lime’s primary environment is a network of mobile low-resource hosts, the main concern of the developers was to introduce these aspects with as low as possible overhead of the original Lime’s model. The security extension is implemented at the level of tuple space access. The privacy extension is implemented as tuple access protection. Moreover, communication between hosts is encrypted to avoid eavesdrop attacks. We shall discuss each level in detail.

Tuple Space protection. In Lime tuple spaces are identified by a name. Two (or more) agents can exchange data by creating a local tuple space with the same name. The underlying Lime system will ensure that the content of each tuple space is visible to the other agent(s). For enforcing security, the tuple space access is protected by a password. An agent will be considered authorized to access a tuple space if it knows both name and password for a given tuple space.

Tuple access protection. Another extension regards the access permissions of tuples. Agents can specify for each tuple that they insert passwords for granting both read and take accesses. To avoid that the storage space of a host is filled up with unaccessible tuples, protected tuples can only be stored in the local tuple space.
Communication protection. Typically, extra-host communication uses unsecured links. For avoiding eavesdropping of messages, each serialized Java object is encrypted using the respective password for accessing a given tuple space. This is however done only when the communication takes place between two (or more) slices of a secure tuple space hosted by different nodes.

**SecSpaces**  SecSpaces [31] provides a calculus for defining secrecy, integrity and availability of data in a tuple space. The authors propose several extension of the tuple space model. The first extension concerns partitioning the tuple space. The partitioning of a tuple space avoids that all the processes have the same view on the data contained in a tuple space. Instead of a physical separation in different tuple spaces, in SecSpaces the tuple space partitioning is achieved through the introduction of a partition field in the tuples. A template can match a tuple in a given partition only if the correct actual value is given in the partition field. A template with a wildcard value in the partition field is considered not valid. This means that a process has to know the name of the partition for accessing the content.

The partitioning of a tuple space is important when it is necessary to authenticate the producers and consumers of data. In particular, the producers are interested in making sure that their data is consumed by trusted consumers. Form the consumers point of view, it is important that the data that they consume was produced by trusted producers. However, the partition field can guarantee this distinction only if the consumer and the producer are the same process.

When the distinction needs to be made between producers and consumers that are different processes then an asymmetric partition field is used. The idea is that for each partition, two corresponding partition field values exist, $K$ and $\overline{K}$. The former is used for introducing the data and the latter for retrieving it. Therefore, if process A posses the value $K$ and value $\overline{K}$ is known only by process B, then it is possible to ensure that the producer and the consumer of the tuple are A and B, respectively.

Another extension regards the distinction between consumers that can only execute read operations and consumers that can only execute take operations. This extension is provided via specified fields in the tuples, called control fields. To be an authorized read consumer, the process has to provide in the template issued by the read operation the exactly value on the read control field of a tuple.

No implementation of this system is described.

**KLAIM**  KLAIM [8] provides privacy by means of encryption. In the framework proposed, a key can be used for encrypting either the data value contained in a field or the complete field (in this case even the type of the field is not visible if the right key is not used).
The model does not provide any access restrictions to the tuple space. This means that encrypted tuples can be retrieved by agents that do not have the right key for decrypting the content. If a tuple is withdrawn from the tuple space by an agent that cannot access it, it is up to that agent to reintroduce the tuple back to the space.

The tuple space API is extended with two operation that execute the decryption process before returning the tuple to the application: ink and readk. If the decryption fails, then the ink operation inserts the tuple back into the space.

Secure Space In Secure Space [90] the approach taken in KLAIM is extended. This framework introduces the notion of lock. A lock is a labeled value that specifies the key that should be used for locking and unlocking the access to a given field. The simplest locks are the symmetric lock where the same key can be used for locking and unlocking a field. Next, an asymmetric lock is provided, where two different keys are necessary for locking and unlocking a field. Usually, a public key is used for locking a field and a private one is used for unlocking it. Finally, the object lock is provided. This lock controls the access to the entire tuple and both symmetric and asymmetric keys are supported.

Multicapabilities in Linda In [89] is described an approach where the multicapability concept is applied to the Linda model. A multicapability is a special capability that refers to a group (multiset) of objects. A permission in a capability grants the holder of the capability certain rights to access the referred object. In a multicapability, a permission grants the holder certain rights to an object of the group to which the multicapability refers to. The set of rights that can be defined in a multicapability is not limited to input and output operations. It can be extend to any type of operations that are appropriate for the system.

The concept of capabilities is a well-known solution for controlling access to objects in a distributed environment. However, capabilities have the big disadvantage that once they have been granted they are difficult to be revoked. This holds as well for multicapabilities. To circumvent this problem, the authors adopt the solution of introducing indirect multicapability objects. A multicapability now refers to the indirection object, which in turn refers to the intended object. The deletion of the indirection object has the effect of removing the multicapability.

5.7 Conclusions

In this chapter the following contribution were made.

First, we presented an extension of the shared data space model where security concerns are treated orthogonally with respect to application functionality and
5.7 Conclusions

data distribution concern. Action related to security are carried out transparently to the applications.

Second, we provided a mean for customizing security policies per tuple type. This makes our approach unique compared to other works where a system-wide approach is used.

Third, the framework has a suite of security mechanisms that are used for building security policies. This suite of mechanisms can be extended with new mechanisms.

Four, the framework provides a simple mechanism for supporting security in a context-aware manner. The activation of a specific security mechanisms is based on either the capabilities of the device that is executing such a mechanism or on the context in which the system is deployed.

As proof-of-concept, we implemented a prototype of GSpace with the security extensions. For proving the feasibility of our approach, in collaboration with Philips Research, we built an implementation of a Personal Health Care System based on GSpace. The orthogonality that our framework offers facilitates the dealing with security concerns. Moreover, adding new security mechanisms did not require any invasive modifications, proving how extensible our framework is.
Chapter 6

Combining Concerns
6.1 Introduction

In the previous chapters we showed that in GSpace extra-functional concerns can be treated separately from the application functionality. In each chapter we discussed an architecture where one specific extra-functional concern is dealt with. Our research shows that extra-functional concerns can successfully be treated orthogonally with respect to application functionality. Additionally, applying the SoC principle allows us to implement an architecture where dynamic adaptation is possible. By dynamically adapting a specific concern to the actual situation our middleware is able to increase the overall performance of the system. Naturally, the question that arises is: what happens when multiple extra-functional concerns have to be dealt with at the same time?

Clearly, when dealing with multiple extra-functional concerns at the same time interactions between separated concerns may occur. The nature of such interactions may be different depending on which concerns are interacting. In some cases, these interactions could lead to sub-optimal utilizations of the resources of the system. In other cases, conflicts may arise that require a more careful handling.

In this chapter we discuss our thoughts on an architecture that can accommodate multiple concerns in the same fashion as described in the previous chapters. In particular, we concentrate on the interactions that could occur when several extra-functional concerns are taken into account.

This chapter is organized as follows. In Section 6.2 we list and motivate the requirements that our multi-concern architecture has to satisfy. Section 6.3 focuses on the interactions that could happen among different concerns. In this section we also provide methods to deal with such interactions. Section 6.4 sketches an architecture that accommodates multiple concerns. We conclude in Section 6.5

6.2 Requirements for a Multi-Concern Architecture

In this section, we list and discuss the requirements that a successful design must satisfy. The requirements are as follows:

1. Separation of Concerns: our main goal remains the separation of application functionality from extra-functional concerns.

2. Transparency of Interweaving: the complexity of the interweaving process for dealing with multiple concerns must be shielded from the application level. Thus the application developer can concentrate on the design of the application without caring about blending functionality with other concerns.
3. Extendable Set of Policies: adding or modifying policies for a specific concern must not affect policies for other concerns.

4. Multi-Concern Adaptability: the architecture must support the adaptation of multiple concerns.

The first and second requirements are the foundations of our research and have been already investigated in the previous chapters of this thesis.

The third requirement is more concerned about the design of the architecture. For instance, we could design an architecture where each policy takes care of availability, performance, and security at once. The code of such a policy would be intertwined with details about different concerns that may turn the writing of such a policy into a nightmare for the developer. Although such a policy is specified outside the code of the application, it violates the SoC principle.

An alternative design choice that is more in line with the SoC principle would be the following. Each policy should deal with a specific concern. However, each supported concern should have in the architecture a specific concern subsystem. In each specific concern subsystem, a concern manager decides which policy should be applied to tuples. The policy is selected from a repository in the middleware. Ideally, each concern subsystem should have its own repository of policies. Such a design improves the development and maintainability of policies. Now a developer who wants to design a new policy for security, for instance, does not have to care about performance and availability concerns.

However, such a design introduces the issue of how to deal with the interactions between policies for different concerns. As we will see later, these interactions can be of two different natures and need to be addressed in different ways.

Although we believe that our architecture covers an extensive set of extra-functional concerns for distributed systems, it might be the case that in the future a new extra-functional concern must be supported. This would require the introduction of the specific concern-subsystem and possibly some modifications to accommodate the dependencies introduced by the new concern. However, this design of the architecture guarantees that there is no need for changing any of other concern-subsystems already present in the architecture.

The fourth requirement focuses on the realization of a method that automatically combines different policies that satisfy the requirements of each concern and that minimizes the overall costs. Naturally, policies for different concerns are going to influence each other during the execution of operations. For instance, encrypting and decrypting tuples during communications for security affects the latency for the execution of an operation, impacting the performance of a distribution policy.

Realizing such a method requires a mechanism for evaluating policies such that:

- it is aware of the interactions between policies for different concerns,
it should compute a solution in reasonable time.

In the next section, we will analyze the nature of interactions between concerns and discuss how to deal with such interactions.

6.3 Concern Interactions

In this section, we discuss how different concerns interact with each other. In particular, we want to analyze the different types of interaction and how we can deal with such interactions in our architecture.

In our architecture, each concern is dealt with in isolation with respect to the others. From the specification of the policy that has to be used to the implementation of the specific mechanism that implements such a policy, concerns are orthogonal to each others. Ideally, such orthogonality should be maintained during execution. In reality, when a policy for one concern is executed it invariably interacts with the execution of other policies.

In the following, we analyze the possible interactions between availability, performance, and security concerns. We first concentrate on the interactions between availability and performance. Afterward, we concentrate on how availability and security interact.

6.3.1 Availability and Performance Interactions

Our framework allows the specification of policies for availability and performance concerns in isolation to each other (and both in isolation to application functionality). This has the advantage for developers to be able to modularize the development of code for availability and for performance in well-specified and separated units. This allows the developers to concentrate on problems related to concerns once at a time, increasing the understandability and efficiency of their code.

A policy for a concern specifies which mechanism should be activated per tuple type. For instance, in the case of availability, mechanisms are distribution strategies that replicate tuples on several nodes for statistically guaranteeing that tuples are still available even in the case that a node crashes. In the case of performance, distribution strategies are used for strategically placing tuples to reduce application latency.

Although the framework separates availability and performance concerns, both concerns leverage on the same type of mechanisms, that is, distribution strategies. To avoid introducing dependencies between concerns inside the framework (to meet our requirements 1 and 3), mechanisms for different concerns are not aware of the
6.3 Concern Interactions

This isolation could lead to a sub-optimal usage of resources. For instance, when a `put` operation is executed a mechanism for availability and a mechanism for performance are activated. These mechanisms could request the same nodes to store the same tuple instance twice. It should be noted that this interaction of mechanisms does not violate the semantics of the space operations. When a `take` operation is executed, both mechanisms will be activated removing both tuple instances. As for `read` operations, the presence of two instances of the same tuple does not affect the semantics of the operation.

However, this modus operandi of the mechanisms has an effect on the resource usage. Inserting the same tuple instance twice on the same nodes requires twice as much memory and bandwidth usage without any further increasing availability or performance.

One possible way to handle such situations is the following. Let us consider a combination of two policies (one for availability and one for performance) that results in a non-optimal usage of resources as a *sub-optimal combination*. This combination is such that the constraints for availability and performance are satisfied without optimization of resource usage. Our problem is shifted to finding a combination of policies such that those constraints are still satisfied and the resource usage is minimized. We refer to such a combination as the *optimal combination*.

To find the optimal combination we resort to our adaptation mechanism. In Chapters 3 and 4, we saw how the adaptation mechanism could be used for automatically finding the optimal policy to be applied to a tuple type. By using a dedicated cost function that combines relevant metrics, the mechanism was used for a specific single concern. For this chapter, we are interested in evaluating combinations of availability and performance policies.

Let $c(a, p)$ be a combination of availability policy $a$ and performance policy $p$. To evaluate such a combination we propose the use of a *general cost function*, indicated as $CF_G$, defined as follows:

$$ CF_G(c(a, p)) = w_1 \cdot CF_A(a) + w_2 \cdot CF_P(p) \quad (6.1) $$

where $CF_A$ and $CF_P$ are the cost functions for availability and performance, respectively. For completeness, we recall the definitions of such functions below:

$$ CF_A(a) = w_1' \cdot bu_a + w_2' \cdot mu_a + w_3' \cdot da_a \quad (6.2) $$

$$ CF_P(p) = w_1'' \cdot rl_p + w_2'' \cdot tl_p + w_3'' \cdot bu_p + w_4'' \cdot mu_p \quad (6.3) $$

where $\sum w'_i = 1$ with $w'_i \geq 0$, and $\sum w''_i = 1$ with $w''_i \geq 0$.

The weights $w_i$ are used for controlling the contribution of each concern. If the application developer is more interested in availability, he could increase the value
of the respective weight to drive the decision of the mechanism towards selecting a combination with more emphasis on the availability policy.

This method minimizes resource usage, such as memory and bandwidth, for both availability and performance policies. At the same time, availability and performance are maximized. This method tries to avoid the selection of a sub-optimal combination. If a better combination exists, this method guarantees that it will find it. In fact, the impact of duplicating tuples will be measured by the memory and bandwidth metrics of both \( CF_A \) and \( CF_P \).

What makes this method appealing is that the input from the application developer is kept to a minimum. The method autonomously evaluates and selects the best combination requiring from the developer only the choice of the weights. However, finding the optimal combination requires to evaluate all possible combinations of availability and performance policies. The complexity of such method is \( O(n \times m) \), where \( n \) is the number of availability policies and \( m \) that of performance.

It is possible to do better with the use of some heuristics. A very simple one could be the following. Availability and performance policies are evaluated separately by the respective cost functions. Each cost function will output the optimal policy for the respective concern, indicated as \( a_{opt} \) for availability and \( p_{opt} \) for performance. The optimal combination then would be \( c(a_{opt}, p_{opt}) \). The complexity of this heuristic is \( O(n + m) \). We suspect that this heuristic will select optimal combinations as long as the interactions between performance and availability are at a minimum. As a matter of fact, this heuristic selects the optimal policies for each concern separately. Thus, if there are interactions between these concerns this heuristic is not able to take them into account.

A variant of the previous heuristic could be the following. The heuristic uses a concern as a pivot. Let us assume that the pivot concern is availability. Using the cost function of availability, the heuristic selects the optimal policy for availability, \( a_{opt} \). Using the general cost function, the heuristic evaluates all combinations in the form of \( c(a_{opt}, p) \), where \( a_{opt} \) is fixed and \( p \) is one of the performance policies. The complexity of this heuristic is \( O(n + m) \). We suspect that this heuristic will select optimal combinations as long as the interactions between performance and availability are at a minimum. As a matter of fact, this heuristic selects the optimal policies for each concern separately. Thus, if there are interactions between these concerns this heuristic is not able to take them into account.

Another heuristic could be the following. The policies of each concern are ranked by using the respective cost functions. After the ranking, the general cost function is used for combining the first \( x \) best policies of both concerns. The complexity of this heuristics is \( O(n + m + x^2) \): \( O(n) \) for ranking the policies for availability; \( O(m) \) for ranking the policies for performance; finally \( O(x^2) \) for finding the optimal combination.

These are just a few examples of heuristics. A heuristic offers the advantage of hav-
6.3 Concern Interactions

Concern interactions may lead to lower complexity with respect to a complete evaluation of all combinations. However, this lower complexity could compromise the precision of the selection. The higher the precision is, the closer the selection is to the optimal combination. By comparing the selection of a heuristic with that of the complete method, we can quantify the precision of that heuristic. Thus the problem is shifted to finding a heuristic that balances a high precision with a low complexity.

6.3.2 Availability and Security Interactions

In this section, we analyze the interaction of availability and security concerns. In particular, we are interested in how mechanisms used for availability could interact with security mechanisms. Since performance uses the same type of mechanisms as availability, the following analysis holds also for the interaction of performance and security.

Availability uses a different type of mechanism than security. Mechanisms in availability are distribution strategies that decide where to place tuples for providing a given availability. At the same time, the cost of maintaining such availability level should be kept at minimum.

Security is a more pervasive concern. Making a system secure (with respect to a threat model) requires taking several precautions scattered across the entire system. For our framework, security mechanisms are concerned with:

- authenticating entities that should perform operations on tuples
- authorizing the operations on tuples
- enforcing privacy of tuples and their content.

The type of interaction between security and availability mechanisms can be explained by the following example. Let us assume that the system is deployed in the environment depicted in Figure 6.1. In the environment, 3 nodes are provided with different characteristics regarding availability and security. Node A provides a high availability but it is rated as not being secure. On the other hand, nodes B and C together provide a lower availability but higher security. The optimal policy for availability would be to store tuples on node A. However, for security constraints such a node cannot be selected.

This type of conflicts can be handled in two ways: fully automated or with user intervention. In fully automated mode, the system is in charge of taking the decision whether or not an operation that violates security constraints should be executed. For instance, per default no security constraints can be violated. In this case, the system has to resort to another distribution strategy that is sub-optimal but that does not violate security. Although this method is fully autonomous, it could lead to a sub-optimal usage of resources.
When the user is involved in the loop, whenever a conflict arises the user is asked to provide guidance for solving the conflict. The user, based on some knowledge, could temporarily allow the violations of security constraints for the sake of optimal resource usage.

Finally, although availability and security concerns are dealt with by different types of mechanisms, during execution these mechanisms can influence each other. For instance, the encryption of a tuple during the execution of an operation influences the overall latency of the operation. However, this is the price to pay for all policies when security is taken into account.

### 6.4 Design for a Multi-Concern Architecture

In this section, we introduce a design for a more flexible architecture that satisfies the requirements given in Section 6.2. The flexibility concerns the capacity of extending the architecture with new managers minimizing the impact (in terms of changes) to other managers already present in the architecture.

Figure 6.2 sketches the design of the proposed architecture and in the following we describe the task of each module.

- **Controller.** This module receives the requests from applications. Requests are dispatched to the managers of each subsystem.

- **Concern Subsystem.** For each concern that our framework supports there is a dedicated subsystem in the architecture. Each subsystem contains the following modules:
Figure 6.2 Multi-concern Architecture for dealing with several concerns.
– **Concern Manager.** A concern manager has to identify which policy has to be activated in function of the type of operation and type of tuple.

– **Policy Repository.** A repository contains a set of policies dedicated to a specific concern. Repositories receive directives from the respective managers for activating policies.

- **Policy Activator.** This module receives policies from the repository of each subsystem and activates them.

- **Monitors.** For each concern, there is a monitor that collects information on the execution of each policy. Each monitor has to be considered as an array of sensor dedicated to the collection of data for a specific concern.

- **Adaptation Manager.** The Adaptation Manager is responsible for collecting the data from the monitors and combine them to get a quantification of the system performance. This quantification is achieved by means of the general cost function ($CF_G$) 6.1. Once the $CF_G$ provides the optimal combination of policies for availability and performance, the Adaptation Manager has to check whether such combination is in conflict with the security constrains. After possible conflicts have been solved, the Adaptation Manager informs the managers about the new combination that has to be activated. The managers then switch to the required policies.

### 6.5 Summary

In this chapter, we proposed an architecture for a multi-concerns framework, where availability, performance and security concerns can be dealt with at the same time. The architecture is such that each concern is isolated from other concerns. We also analyzed interactions among the concerns that are currently supported. We differentiated two types of interactions: those that lead to sub-optimal solutions and those that lead to conflicts. For each type, we proposed methods that could be used for dealing with them.

We conclude that our approach is feasible for integrating multiple concerns while fully adhering to the SoC principle. Moreover, our approach is able to deal and solve interactions among those supported concerns in an automated fashion. This is a crucial contribution of our research that make it standing out from AOP approaches where the interactions between aspects is still an open problem. However, we realize that the work presented in this chapter needs to be substantiated by experimental proof. We consider this as the main topic of our future research.
Chapter 7

Conclusions
In this thesis, we discussed the design of a framework for assisting application designers in development and deployment of distributed applications. The key features of our framework are:

- **Separation of concerns:** our framework provides a clear separation between the basic functionality of an application from its extra-functional concerns, such as performance, availability and security. This clear separation allows the application developer to specify application functionality in isolation from the other concerns. This increases software-quality properties such as reusability, traceability, and evolution of application components and code used for dealing with extra-functional concerns.

- **Run-time adaptation:** implementing concerns separately enables our framework to dynamically adapt extra-functional concerns to the actual needs of the application. In our framework, each extra-functional concern is dealt with via a library of concern-specific mechanisms. After the application is deployed, the system monitors the application behavior and the deployment environment. Whenever the monitored parameters change, the system tries to adapt to the changes by switching to a suitable strategy.

To validate our research, we implemented our framework as a middleware in the form of a shared data space. This implementation was named GSpace. GSpace is a proof-of-concept and was used for conducting experiments and for building case study applications. The findings of the experiments and measurements conducted using GSpace can be summarized as follows:

- The support of multiple distribution or availability policies outperformed any fixed-policy solutions.

- The adaptation mechanism embedded in GSpace allows the system to dynamical reconfigure distribution and availability policies to meet the requirements of the application over a changeable environment.

- The overhead of the adaptive mechanism is small compared to the gain (either in terms of performance or availability) that the overall system obtains.

The validation of separating security was performed by applying the architecture for designing an industrial prototype. In particular, GSpace was used in collaboration with Philips Research Laboratories for building a prototype of an application for Personal Health Care. The aim of this case study was to support security transparently to the application. Our major findings are of a software engineering nature. Our framework shows that such a pervasive concern can be treated orthogonally with respect to the application functionality and other extra-functional concerns. In this way, the writing of the functionality code of the Personal Health
Care components was rather simple. This is mainly due to the fact that all the
details about security were removed from the application code.

As future research directions we see three main areas of interest:

1. GSpace deployment over large-scale and mobile ad-hoc networks (MANET):
   GSpace was initially developed for LAN environments where the primary use
   would have been for in-home networks of consumer devices. As for MANETs,
   we believe that GSpace can easily cope with the dynamic nature intrinsic to
   this environment. Due to the transparency of data distribution that is offered
   to the application, GSpace can minimize the impact of dynamic engagements
   and disengagements of devices. However, some redesign is necessary for the
   adaptation mechanism. The current approach of a centralized adaptation
   master that takes decisions for the entire system is not feasible in a MANET
   environment.

2. Support of an adaptation mechanism for security concern: GSpace supports
   adaptation for performance and availability concerns. The goal of the adap-
   tation mechanisms is to try to provide a certain quality of service (such
   as availability) while minimizing the costs. To achieve this, the adapta-
   tion mechanism uses a cost function that combines different metrics that
   measure the benefits and costs of a mechanism used for a specific concern.
   This approach is applicable whenever it is possible to use some metrics that
   quantify a given concern. As a matter of fact, security is difficult to quantify.
   Providing some meaningful measurements of how secure a mechanism is, is
   somehow artificial. This limits the use of the adaptation mechanism as a
   mean for optimizing costs. However, adaptation can be used for a different
   purpose. The adaptation mechanism can decide to change level of security
   in response to changes in the environment. For instance, if the mechanism
   detects that data is communicated over an unsecure link (such as a public
   wireless network) then the mechanism can automatically activate an encryp-
   tion mechanism to protect the data. This shifts the focus of the adaptation
   from an efficiency perspective to a more context-aware perspective.

3. Realization of the multi-concern architecture: Chapter 6 focuses on the com-
   bination of multiple concerns in a single architecture. Although we discussed
   how such an architecture could be realized, it would be interesting to im-
   plement such an architecture. More interesting would be the research of
   heuristics for finding the proper combinations of mechanisms to be used in
   the adaptation mechanism.
References


135
136 References


References


References


Summary

With its decoupling of processes in space and time, the shared data space model has proven to be well-suited for developing distributed component-based systems. Decoupling in space means that components do not need to know their respective locations in order to establish communication. Decoupling in time means that components are able to communicate without the need to be active at the same time.

Another key feature of the shared data space model is that the model supports a clear separation between the computational part of an application from its coordination part. This allows the application developer to define the functionality of an application (the computational part) without having to consider details about the interaction with other components in the environment where the application is going to be deployed (the coordination part). In the coordination part these details can be dealt with separately and adapted to the specific requirements of the assembled system.

This is in line with the principle of separation of concerns. According to this principle, the design of software is improved by separating the code for the basic functionality of an application from the code for dealing with extra-functional concerns, such as distribution, availability, security and timing.

Although the shared data space model is well suited for supporting separation of concerns, it is our opinion that most of the research on the shared data space model has failed in exploiting this feature. This has the following consequences:

- A plethora of implementations has been proposed all with fixed solutions for specific environments. This has the drawback of limiting the genericity and flexibility of those systems.
- The functional and extra-functional aspects are still interwoven in the code that defines the application components. As drawback, the advantages of the separation of concerns are lost.

In this thesis, we address how shared data spaces can support separation of con-
cerns. In particular, we present an architecture that allows developers to define performance, availability and security requirements separately from the application functionality.

Additionally, and this is a unique feature of our approach, the architecture enables the continuous monitoring of the behavior of application components and hardware, and based on this, the shared data space can at run time switch to a policy for realizing extra-functional properties that best fits the needs of the system.

In this thesis, we describe our approach, a design of a prototype implementation and its experimental evaluation.
Vanwege zijn ontkoppeling van processen in ruimte en tijd heeft het shared data space model bewezen een geschikte oplossing te zijn voor het ontwikkelen van gedistribueerde componentgebaseerde systemen. Ontkoppeling in ruimte betekent dat componenten kunnen communiceren zonder dat zij elkaars locatie hoeven te kennen. Ontkoppeling in tijd betekent dat componenten kunnen communiceren zonder dat ze gelijktijdig actief zijn.

Een andere belangrijk aspect van het shared data space model is dat het ondersteuning biedt voor een strikte scheiding tussen het computationele gedeelte van een applicatie en het coördinatie gedeelte. Dit maakt het mogelijk voor applicatieontwikkelaars om de functionaliteit van een applicatie (het computationele gedeelte) te definiëren zonder rekening te houden met de afstemming met andere applicaties die in dezelfde omgeving worden uitgevoerd (het coördinatie gedeelte). In het coördinatie gedeelte kunnen deze details worden afgehandeld en worden aangepast aan de specifieke eisen van het gehele systeem.

Dit volgt het principe van belangenscheiding. Volgens dit principe wordt het ontwerp van software verbeterd wanneer de basis functionaliteit van een applicatie gescheiden wordt van de extrafunctionele aspecten zoals distributie, beschikbaarheid, beveiliging en timing. Deze scheiding vergroot de toepasbaarheid van componenten en vereenvoudigt het onderhoud van componenten.

Hoewel het shared data space model zeer geschikt is voor het ondersteunen van belangenscheiding, zijn wij van mening dat het meeste onderzoek naar het shared data space model faalt in het exploiteren van deze eigenschap. Dit heeft de volgende consequenties:

- Er is een veelvoud aan implementaties is voorgesteld die allemaal toegespitst zijn op een specifieke omgeving. Hierdoor wordt de algemene toepasbaarheid en flexibiliteit van deze systemen beperkt.

- De functionele en extrafunctionele aspecten zijn verweven in de code van de applicatie-componenten. Hierdoor worden de voordelen van belangenscheiding niet gerealiseerd.
In dit proefschrift bekijken we hoe een architectuur voor shared data spaces ondersteuning kan bieden voor belangenscheiding. In het bijzonder, presenteren we een ontwerp dat ontwikkelaars in staat stelt om performance, beschikbaarheid en beveiliging los van de applicatie functionaliteit te definiëren.

Verder, en dit is uniek in onze oplossing, maakt de architectuur het mogelijk om toezicht te houden op het gedrag van de applicatiecomponenten en hardware, en om op basis hiervan het beleid voor het realiseren van extra-functionele eigenschappen tijdens run-time te veranderen zodat dit beter past bij de doelen van het systeem.

In dit proefschrift beschrijven we onze aanpak, het ontwerp van een prototype implementatie en een experimentele evaluatie.
Curriculum Vitae

Giovanni Russello was born on the 26th of January 1976 in Gela, Italy. He received his Master degree (summa cum laude), in Computer Science (Technical field), from University of Catania, Italy in October 2000. Since May 2001, he has been a Ph.D. student within the Computer Science Department of the Technology University of Eindhoven (TU/e) in Eindhoven, The Netherlands. His research was funded by NWO (The Dutch Organization for Scientific Research) within the project SACC: Software Architecture = Components + Coordination. Giovanni carried out the last year of his research at the Philips Research Laboratories, Eindhoven, The Netherlands. The results of his research have been positively accepted among the research community leading to several publications. This thesis represents the summa of those publications.

As for April 2006, Giovanni moved to London, U.K. to join the Department of Computing at Imperial College London as an Associate Researcher.
Titles in the IPA Dissertation Series


A.M. Geerling. *Transformational Development of Data-Parallel Algorithms*. Faculty of Mathematics and Computer Science, KUN. 1996-02

P.M. Achten. *Interactive Functional Programs: Models, Methods, and Implementation*. Faculty of Mathematics and Computer Science, KUN. 1996-03


D. Turi. *Functorial Operational Semantics and its Denotational Dual*. Faculty of Mathematics and Computer Science, VUA. 1996-09


N.W.A. Arends. *A Systems Engineering Specification Formalism*. Faculty of Mechanical Engineering, TUE. 1996-11

P. Severi de Santiago. *Normalisation in Lambda Calculus and its Relation to Type Inference*. Faculty of Mathematics and Computing Science, TUE. 1996-12


B.L.E. de Fluiter. *Algorithms for Graphs of Small Treewidth*. Faculty of Mathematics and Computer Science, UU. 1997-01


F.A.M. van den Beuken. *A Functional Approach to Syntax and Typing*. Faculty of Mathematics and Informatics, KUN. 1997-07

A.W. Heerink. *Ins and Outs in Refusal Testing*. Faculty of Computer Science, UT. 1998-01


J. Verriet. *Scheduling with Communication for Multiprocessor Computation*. Faculty of Mathematics and Computer Science, UU. 1998-03


E. Voermans. Inductive Datatypes with Laws and Subtyping – A Relational Model. Faculty of Mathematics and Computing Science, TUE. 1999-01

H. ter Doest. Towards Probabilistic Unification-based Parsing. Faculty of Computer Science, UT. 1999-02


C.H.M. van Kemenade. Recombinative Evolutionary Search. Faculty of Mathematics and Natural Sciences, UL. 1999-04

E.I. Barakova. Learning Reliability: a Study on Indecisiveness in Sample Selection. Faculty of Mathematics and Natural Sciences, RUG. 1999-05


J.P. Warners. Nonlinear approaches to satisfiability problems. Faculty of Mathematics and Computing Science, TUE. 1999-08


P.R. D’Argenio. Algebras and Automata for Timed and Stochastic Systems. Faculty of Computer Science, UT. 1999-10

G. Fábián. A Language and Simulator for Hybrid Systems. Faculty of Mechanical Engineering, TUE. 1999-11


R. Schiefer. Viper, A Visualisation Tool for Parallel Program Construction. Faculty of Mathematics and Computing Science, TUE. 1999-15


T.E.J. Vos. UNITY in Diversity. A stratified approach to the verification of distributed algorithms. Faculty of Mathematics and Computer Science, UU. 2000-02


P.H.F.M. Verhoeven. The Design of the MathSpad Editor. Faculty of Mathematics and Computing Science, TUE. 2000-05

J. Fey. Design of a Fruit Juice Blending and Packaging Plant. Faculty of Mechanical Engineering, TUE. 2000-06


P.A. Olivier. A Framework for Debugging Heterogeneous Applications. Faculty of Natu-
E. Saaman. *Another Formal Specification Language*. Faculty of Mathematics and Natural Sciences, RUG. 2000-10

M. Jelasity. *The Shape of Evolutionary Search Discovering and Representing Search Space Structure*. Faculty of Mathematics and Natural Sciences, UL. 2001-01

R. Ahn. *Agents, Objects and Events a computational approach to knowledge, observation and communication*. Faculty of Mathematics and Computing Science, TU/e. 2001-02

M. Huisman. *Reasoning about Java programs in higher order logic using PVS and Isabelle*. Faculty of Science, KUN. 2001-03


S.C.C. Blom. *Term Graph Rewriting: syntax and semantics*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2001-05

R. van Liere. *Studies in Interactive Visualization*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03


J. Hage. *Structural Aspects of Switching Classes*. Faculty of Mathematics and Natural Sciences, UL. 2001-08

M.H. Lamers. *Neural Networks for Analysis of Data in Environmental Epidemiology: A Case-study into Acute Effects of Air Pollution Episodes*. Faculty of Mathematics and Natural Sciences, UL. 2001-09

T.C. Ruys. *Towards Effective Model Checking*. Faculty of Computer Science, UT. 2001-10

D. Chklaev. *Mechanical verification of concurrency control and recovery protocols*. Faculty of Mathematics and Computing Science, TU/e. 2001-11


M.C. van Wezel. *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects*. Faculty of Mathematics and Natural Sciences, UL. 2002-01

V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02


S.P. Luttik. *Choice Quantification in Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04


N. van Vugt. *Models of Molecular Computing*. Faculty of Mathematics and Natural Sciences, UL. 2002-07
A. Fehnker. Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems. Faculty of Science, Mathematics and Computer Science, KUN. 2002-08

R. van Stee. On-line Scheduling and Bin Packing. Faculty of Mathematics and Natural Sciences, UL. 2002-09


M.B. van der Zwaag. Models and Logics for Process Algebra. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-11


J.I. van Hemert. Applying Evolutionary Computation to Constraint Satisfaction and Data Mining. Faculty of Mathematics and Natural Sciences, UL. 2002-14


Y.S. Usenko. Linearization in μCRL. Faculty of Mathematics and Computer Science, TU/e. 2002-16

J.J.D. Aerts. Random Redundant Storage for Video on Demand. Faculty of Mathematics and Computer Science, TU/e. 2003-01

M. de Jonge. To Reuse or To Be Reused: Techniques for component composition and construction. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02


S.M. Bohte. Spiking Neural Networks. Faculty of Mathematics and Natural Sciences, UL. 2003-04


S.V. Nedea. Analysis and Simulations of Catalytic Reactions. Faculty of Mathematics and Computer Science, TU/e. 2003-06


H.P. Benz. Casual Multimedia Process Annotation – CoMPAs. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-08


M.H. ter Beek. Team Automata – A Formal Approach to the Modeling of Collaboration Between System Components. Faculty of Mathematics and Natural Sciences, UL. 2003-10

D.J.P. Leijen. The λ Abroad – A Functional Approach to Software Components. Faculty of Mathematics and Computer Science, UU. 2003-11


G.I. Jojgov. Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving. Faculty of Mathematics and Computer Science, TU/e. 2004-02

P. Frisco. Theory of Molecular Computing – Splicing and Membrane systems. Faculty of
S. Maneth. Models of Tree Translation. Faculty of Mathematics and Natural Sciences, UL. 2004-03

Y. Qian. Data Synchronization and Browsing for Home Environments. Faculty of Mathematics and Computer Science and Faculty of Industrial Design, TU/e. 2004-05


L. Cruz-Filipe. Constructive Real Analysis: a Type-Theoretical Formalization and Applications. Faculty of Science, Mathematics and Computer Science, KUN. 2004-07

E.H. Gerding. Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies, and Business Applications. Faculty of Technology Management, TU/e. 2004-08

N. Goga. Control and Selection Techniques for the Automated Testing of Reactive Systems. Faculty of Mathematics and Computer Science, TU/e. 2004-09


J. Pang. Formal Verification of Distributed Systems. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-14

F. Alkemade. Evolutionary Agent-Based Economics. Faculty of Technology Management, TU/e. 2004-15

E.O. Dijk. Indoor Ultrasonic Position Estimation Using a Single Base Station. Faculty of Mathematics and Computer Science, TU/e. 2004-16

S.M. Orzan. On Distributed Verification and Verified Distribution. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-17


P.J.L. Cuijpers. Hybrid Process Algebra. Faculty of Mathematics and Computer Science, TU/e. 2004-20

N.J.M. van den Nieuwelaar. Supervisory Machine Control by Predictive-Reactive Scheduling. Faculty of Mechanical Engineering, TU/e. 2004-21

E. Ábrahám. An Assertional Proof System for Multithreaded Java - Theory and Tool Support-. Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. Modeling and Remodeling in Bone Tissue. Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. Experiments in Rights Control - Expression and Enforcement. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. Design and Verification of Lock-free Parallel Algorithms. Faculty of Mathematics and Computing Sciences, RUG. 2005-04
H.M.A. van Beek. Specification and Analysis of Internet Applications. Faculty of Mathematics and Computer Science, TU/e. 2005-05


I. Kurtev. Adaptability of Model Transformations. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. Computational Aspects of Treewidth - Lower Bounds and Network Reliability. Faculty of Science, UU. 2005-09

O. Tveretina. Decision Procedures for Equality Logic with Uninterpreted Functions. Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. Evolution of Finite Populations in Dynamic Environments. Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. Data Mining using Genetic Programming: Classification and Symbolic Regression. Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. Top Quality Type Error Messages. Faculty of Science, UU. 2005-13

G.F. Frehse. Compositional Verification of Hybrid Systems using Simulation Relations. Faculty of Science, Mathematics and Computer Science, RU. 2005-14


T. Gelsema. Effective Models for the Structure of pi-Calculus Processes with Replication. Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. Composing Constraint Solvers. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18


M. Valero Espada. Modal Abstraction and Replication of Processes with Data. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. Stepping through Haskell. Faculty of Science, UU. 2005-21


E. Dolstra. The Purely Functional Software Deployment Model. Faculty of Science, UU. 2006-01


P.R.A. Verbaan. The Computational Complexity of Evolving Systems. Faculty of Science, UU. 2006-03


M. Kyas. Verifying OCL Specifications of UML Models: Tool Support and Compositionality. Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. Model Checking Timed Automata - Techniques and Applications. Fac-
J. Ketema. *Böhm-Like Trees for Rewriting.* Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs.* Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations.* Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data.* Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space.* Faculty of Mathematics and Computer Science, TU/e. 2006-11
Appendix A

Alternative Strategy for Adaptation
A.1 Introduction

Ideally, the changing of policy during an adaptation phase should be executed instantaneously on all nodes. Achieving atomicity in a distributed system requires some extra effort. For simplicity, in the current implementation before switching policy no operations are allowed to be executed. Simply put, it is like stretching a single point in time to gain enough time for adapting policy in all nodes. However, this has the disadvantage of making the system unusable during the adaptation. If the adaptation is not executed instantaneously, operations on different nodes could be executed according to different strategies. Although this issue does not affect the integrity of the system, the consistency of the model could be compromised. As a matter of fact, tuples might be misplaced and retrieval operations could search for a matching tuple in the wrong location. This leads to the case that a read or take operation fails to retrieve a matching tuple even if the tuple is in the system, only not in the right location.

A.2 Modifications

To overcome the problem of halting the system during an adaptation phase, our current approach can be changed to avoid that the system is frozen during adaptation. When the master AM decides that a policy for a tuple type has to be changed, it starts sending message to all the other kernels. However, these messages only inform the other AM which policy should be used next for a giving tuple type. To avoid the problem of misplacing tuple, two modifications are required. The first modification affects the GSpace kernel. Each kernel has to keep track of the adaptation era of each tuple type. Basically, it is a integer that is initially 0. At each policy adaptation for a specific tuple type, the adaptation era is incremented by one. The second modification concerns the Java class Tuple used in GSpace as root class of tuple types defined by applications. This root class contains several fields that are used only by the system for administration. The lookup mechanism ignores these fields during the searching of a matching tuple. Moreover, these fields are not to be accessed by application components. To support the new mechanism the class Tuple is extended with two fields: policyName and adaptationEra. The first field contains the name of the policy according to which the tuple was distributed. This field is also used for templates (since a template is just an ordinary tuple that may contain some wildcards), only here the policy name represents the strategy that is used for searching for a matching tuple. The second field is used for indicating the adaptation era of the policy value associated with the tuple or template. When an application component executes an operation, the argument of the operation, either a tuple or a template, is tagged with policy name and adaptation era of the node\(^1\) where the operation is being served.

When a kernel executes an operation, a policy is in charge of executing the operation according to its strategy. Basically, a policy can either decide to store the complete the operation by executing it on the local Data Space Slice or forward the operation to other kernels. Eventually, the operation is executed on a Data Space Slice. However, before the operation is completed in a Data Space Slice, the policyName of the argument of the operation (either tuple or template) is checked against the values contained in the local kernel. In case that the policy values in the tuple field and in the Policy Table are not the same due to the fact that an adaptation is in course then some actions needs to be taken as formally explained in the following.

\(^1\)In the following we use the term node to indicate the GSpace kernel running on that node
A.3 Notation

Let us assume that the system is adapting from policy \( p \) of era \( n \) to policy \( q \) and era \( n+1 \). For simplicity, we assume that there is a single tuple type in the system, thus we can omit the tuple type from our notation. Policy \( p \) stores and retrieves tuples in and from node \( I \). Instead, policy \( q \) uses node \( J \) for storing and retrieving tuples. The notation \( t(p, n) \) indicates that the tuple \( t \) has policyName set to \( p \) and adaptationEra set to \( n \). Similarly, notation \( templ(p, n) \) indicates that template \( templ \) has policyName set to \( p \) and adaptationEra set to \( n \). Notation \( I(p, n) \) indicates that on node \( I \) the Policy Table has an entry for the tuple type associated with policy \( p \), and the current adaptation era is \( n \). Because the system is not halted during the adaptation, it could be the case that the node where the operation is originated, indicated as \( O \), and the destination node, indicated as \( D \), have different policy values and adaptation eras specified for the tuple type. In particular, the two following cases are relevant to our problem:

1. \( O(p, n) \) and \( D(q, n+1) \): at the time when the operation is executed, the node where the operation is originated has not been updated. Whereas the destination node already received the adaptation message and the policy value and current adaptation era are updated accordingly. Since the operation is executed according to policy \( p \), the destination node of the operation is node \( I \). According to our notation we have \( I(q, n+1) \). As we said, at the time an operation is executed the argument fields (policyName and adaptationEra) are set to the values of the origin node. Thus we have for a tuple \( t(p, n) \) and for a template \( templ(p, n) \). When the request for executing the operation arrives to the destination node \( I \), the fields of the operation argument are compared with the respective fields of node \( I \). In particular, from the era value of the argument field is evicted that the operation was originated from a node that has not been updated yet. Therefore the operation cannot be executed in this node but has to be handled by the node prescribed by policy \( q \). The mechanism proceeds distinguish the following two cases:

   • The operation is a put and the argument is \( t(p, n) \). In this case, the fields are updated with the new values \( q \) and \( n+1 \), respectively. Afterwards, the operation is forwarded to the appropriate node, that according to policy \( q \) is node \( J \). We assume here that node \( J \) has been updated \((J(q, n+1)) \). If not then this falls in case 2 that is treated shortly below. Once the request arrives to node \( J \), the tuple is inserted in its Data Space Slice since the policy and adaptation era values match.

   • The operation is either a read or a take and the argument is \( templ(p, n) \). The template fields are updated with the new values, and the operation is forwarded to node \( J \). We assume here that node \( J \) has been updated to \( J(q, n+1) \). Otherwise, this falls in case 2 that is described below. Once the request arrives in \( J \), the Data Space Slice of node \( J \) is searched for a tuple matching the template.

2. \( O(q, n+1) \) and \( D(p, n) \): this is the dual of the above case. The origin node has been update by the master AM, but not the destination node. When a request arrives to node \( J \), the values of the argument of the operation (either a tuple or a template) are checked with the local values and found different. In particular, either tuples or templates have fields set to \( (q, n+1) \). From the adaptation era value it is evicted that the values of node \( J \) are stale and needs to be updated. The AM of node \( J \) updates the local Policy Table and the adaptation era and starts the transition policy to update all tuples stored in the local Data Space Slice. Whenever the update message from the master AM arrives to node \( J \), no further actions are taken since the node is already updated.

If the system is adapting from a policy that support replication to a policy that does not, extra case needs to be taken to discern between tuples and their replicas. This is done using the replica flag of the class Tuple that is set to true when a tuple is a replica.

Worth noting is that the above protocol can be used for changing the descriptor file during runtime.