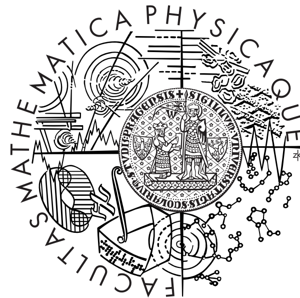
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Václav Remeš

Migration and load-balancing in distributed hierarchical component systems

Department of Distributed and Dependable Systems

Supervisor: RNDr. Petr Hnětynka, Ph.D.

Study program: Computer Science - Software Systems

2010

I would like to thank my supervisor for all his advice and tolerance, my parents for all their support and infinite patience, my girlfriend for her amiability and her sense of humor at 2 a.m. and all the people who gave me useful advice and helped me stay on the right side of sanity.

I hereby state I have written my master thesis singly and with exclusive use of referenced sources. I agree with its lending and publishing.

In Prague, Václav Remeš

Contents

1 Introduction **7**
1.1 Distributed hierarchical component systems . . . 7
1.2 Migration and load balancing . . . 8
1.3 Goal of the thesis . . . 8
1.4 Structure of the thesis . . . 9

2 Background **10**
2.1 Components . . . 10
2.1.1 Component examples . . . 11
2.2 Connectors . . . 12
2.3 SOFA 2 . . . 12
2.3.1 Runtime architecture of SOFA 2 . . . 13
2.3.2 Metamodel . . . 14
2.3.3 Launching a SOFA 2 application . . . 16

3 Implementing migration in a component system **17**
3.1 Migration process in theory . . . 17
3.1.1 Primitive components vs. composed ones . . . 17
3.1.2 Anticipated vs. unanticipated . . . 18
3.1.3 Overview of the migration process . . . 19
3.2 Implementing migration . . . 20
3.2.1 Migrable components in SOFA 2 . . . 20
3.2.2 Stopping the component's jobs . . . 24
3.2.3 Saving the state of a component . . . 26
3.2.4 Re-instantiating the component . . . 28
3.2.5 Reconnecting the components . . . 28
3.2.6 State loading and resuming component's work . . . 34
3.2.7 Finalization . . . 35
3.2.8 Handling error states during migration - reverting back to the original component . . . 35
3.2.9 Possible problems with concurrency . . . 36
3.3 Handover protocol . . . 36
3.3.1 Migration influence on the rest of the application . . . 36

3.3.2 Method invocation - incoming calls during migration 36
3.3.3 Reconnection of messaging connectors 38
3.3.4 Migration of a larger number of components 39
3.3.5 Possible problems of the method 40
3.4 Improvements 40
3.4.1 Preinstantiation and concurrency 42

4 Load balancing 43
4.1 Reasons for load balancing at runtime 43
4.1.1 Resource usage variances 43
4.1.2 Migrating clients to servers when huge load of communication is anticipated 44
4.2 Monitoring system load in a distributed system 44
4.3 Load balancing principles 44
4.3.1 Example 45

5 Prototype implementation 46
5.1 Changes which had to be made to SOFA 2 46
5.1.1 Exclusive usage of migrating connectors 46
5.1.2 Deployment Dock changes 48
5.1.3 Executing migration 49
5.2 Load balancing 49

6 Evaluation 51
6.1 Migration time evaluation 51
6.1.1 Testing environment 51
6.1.2 Results 52

7 Related work 55
7.1 Migration in CORBA 55
7.2 Migration in ProActive 56

8 Conclusion and future work 59
8.1 Summary 59
8.2 Future work 59

Literature 61

A Contents of the enclosed disc 63
A.1 Structure of the disc 63
A.2 Examples howto 63
A.2.1 System requirements 63
A.2.2 Running the examples 64
A.2.3 List of the example Deployment Plans 65

A.2.4 Migration in MConsole 66
A.3 Known issues 67

Title: Migration and load-balancing in distributed hierarchical component systems
Author: Václav Remeš
Department: Department of Distributed and Dependable Systems
Supervisor: RNDr. Petr Hnětynka, Ph.D.
Supervisor's e-mail address: hnetynka@d3s.mff.cuni.cz

Abstract: A well balanced usage of resources is one of the goals of distributed applications. A way to achieve such a balanced usage is by run-time monitoring and migration of components of already executed applications between computers. There are many issues related to migration, from monitoring resources usage till obtaining component state and transferring to a different computer. The goal of this thesis is to design and implement a support for migration and load-balancing of components in the SOFA 2 hierarchical component system.

Keywords: migration, distributed systems, load balancing

Název práce: Migration and load-balancing in distributed hierarchical component systems
Autor: Václav Remeš
Katedra (ústav): Katedra distribuovaných a spolehlivých systémů
Vedoucí diplomové práce: RNDr. Petr Hnětynka, Ph.D.
e-mail vedoucího: hnetynka@d3s.mff.cuni.cz

Abstrakt: Vyvážené využití zdrojů je jedním z cílů distribuovaných aplikací. Jeden ze způsobů, jak dosáhnout vyváženého využití je monitorování běhu aplikací a migrace jejich komponent mezi počítači. Migrace s sebou přináší mnoho otázek, od monitorování prostředků až po získávání stavu komponenty a jeho přenosu na jiný počítač. Cílem této práce je navrhnout a implementovat podporu pro migraci a vyvažování zátěže v hierarchickém komponentovém systému SOFA 2.

Klíčová slova: migrace, distribuované systémy, vyvažování zátěže

Chapter 1

Introduction

Distributed systems are taking a great importance in today's computing. There are many ways to develop a distributed system. They start with creating an in-house protocol and beginning from the lowest level. After that follows the option of using some already available middleware systems like Java RMI, CORBA or .NET Remoting. The highest level approach is the possibility of using some sophisticated frameworks like EJB, CCM, DCOM, SOFA 2, etc., which provide a transparent abstraction over the distribution and communication of the particular parts of software.

1.1 Distributed hierarchical component systems

When designing a large and complicated system, particularly when it is a distributed system, the overall work is very simplified when it can be split into a number of logical parts separated from each other. The parts can then operate as black boxes and communicate with each other only using well defined provided and required interfaces. These black boxes are called components.

If a system is designed using the component approach and the underlying framework allows it, the developers of the application can get many benefits without (or almost without) any effort. First of all, the components can be developed by independent teams of programmers that can focus only on their own parts and just make sure they use the provided and required interfaces of their component correctly. Dividing components into autonomous units makes the verification of their correctness much easier. Components can be distributed among a network of computers, they can be updated and even replaced at runtime and many more.

Last but not least, components can be migrated between the computers which are involved in the system to balance the load of the computers, their cpu, memory, disk, networking usage and many more. These are precisely the topics which are covered by the thesis.

There exists quite a large number of component systems. For example SOFA 2 [4], Fractal [22], EJB [23], DCOM [24] and many more.

1.2 Migration and load balancing

Process migration has been discussed a lot during the 90th's in the context of distributed operating systems. A lot of research was done those days and some distributed operating systems which incorporated process migration at runtime saw the sun, for instance Amoeba [25], MOSIX [27], Sprite [26], etc.

However, the topic was left dead for many years until around 2000 A.D. computer clusters came into fashion, also migration of virtual computers started to be regarded highly profitable and so on. These cases are however a bit different from our research subject since they comprise a very low-level approach at the operating system layer. When implementing migration into a general distributed component system, there is no guarantee that the system has direct access to the memory space or even have a complete knowledge about the state of a component. That's why component systems can take inspiration in distributed operating systems but also have to develop methods of their own when migration is concerned.

The possibility to migrate components between different computers involved in a distributed system brings many advantages. First of all when a computer needs to be shut down, the possibility of just migrating away all components which have been running on it is highly desirable. Another possibility is to have a monitoring system periodically checking the utilization of the computers that the components are running on. With a bit of analysis the system can migrate components to better spread the load on the computers and utilize them more effectively.

Migration can be even used in cases when the system "knows" that two components will be communicating a lot in the following period of time; they can be migrated to the same node to let them communicate directly - without remote calls that are, of course, much slower.

It is clear that adapting the component system's topology at runtime for either load balancing or any other purposes first requires a fully working migration functionality. This thesis therefore focuses mainly on the migration part and covers load balancing only marginally.

1.3 Goal of the thesis

The goal of the thesis is to analyze the possibilities of migration in component systems and to propose a working prototype implementation of migration into the SOFA 2 component system. There are many issues to be solved when wanting to provide a fully functioning component migration. The most notable issues which the thesis should solve are transferring the component's state between different nodes, the ways of leading the component into a reconfigurable state and solving dynamic dependencies of the inter-component calls to prevent deadlocks. Different communication styles also require different approaches when reconnecting the software connectors.

The prototype implementation should come also with a simple system analyzing

the performance of the running applications and balancing the load of the computers by migrating components between them.

The thesis focuses mainly on the migration issues.

1.4 Structure of the thesis

Chapter 2 covers background information needed to comprehend the rest of the thesis.
Chapter 3 aims at describing the problems of migrating components and ways of solving them.
Chapter 4 considers the possibilities of utilizing migration to balance the load of the distributed system.
Chapter 5 covers the prototype implementation which was developed for the SOFA 2 component system as the prototype implementation of the thesis.
Chapter 6 evaluates the work done on the thesis and presents the results of some measurements.
Chapter 7 tries to delimitate the work done for the thesis to other systems which provide the migrating functionality.
Chapter 8 sums up the whole thesis and proposes additional work which could be done in the field.

Chapter 2

Background

In this chapter we focus on the background information needed to comprehend this thesis. The following sections cover the basics of software components and component systems, principles of software connectors and their benefits in distributed systems are taken into account next. The last section contains a brief description of the SOFA 2 component system.

2.1 Components

Before we can start talking about migration in component systems, we should make clear what we understand by the term of "Component".

This is what a component is defined like by the OMG [5]:

> A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment. A component defines its behavior in terms of provided and required interfaces. As such, a component serves as a type whose conformance is defined by these provided and required interfaces (encompassing both their static as well as dynamic semantics).

In this thesis we see components as black boxes that are either capable of executing their own business code (we call them primitive components) or contain other components (these are called composed components). A component should contain processes which are strongly related with each other and it is recommended for the components to have coarse-grained interfaces to follow the high-coupling and low-cohesion rule.

These black boxes expose some provided interfaces to which other components are connected using their required interfaces. The component itself should not be aware of what is on the opposite side of its interfaces nor should its internals be concerned too much with the underlying middleware (of course this is just a theoretical recommendation, it should for example know whether the communication is synchronous or asynchronous).

Note that the components are forbidden to have other means of communication than their interfaces. They also cannot share any global variables and so on.

2.1.1 Component examples

Figure 2.1: Example of primitive components

Figure 2.1 shows an example of two components, Component A is connected through its required interface "foo" to Component B's provided interface "foo".

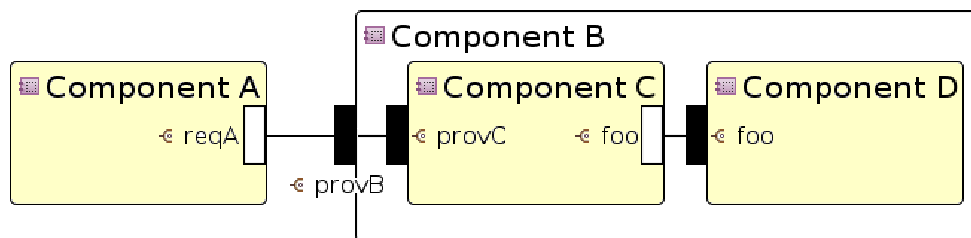


Figure 2.2: Example of composed components

In Figure 2.2 there can be seen an example of composed components (in the figure primitive components have a yellow background color). Primitive component A is connected to a composed component B which contains components C and D.

The question is how composed components are connected with their subcomponents. Since a normal connection stands between a required and a provided interface, it cannot be used in the case of composed components because the containing component must connect its required interfaces to its subcomponent's required interfaces and the same goes with provided interfaces.

When a subcomponent's provided interface is connected to its containing component we say that the provided interface is *subsumed* (the connection is thus called *subsumption*). Composed component's required interfaces are said to be *delegated* to its subcomponents (and so the connection is called *delegation*).

For a more illuminating example see Figure 2.3. In this figure composed component A's provided interface *provA* is *subsumed* to component B's provided interface *provB*. Between component B and component C there is a normal binding between the interfaces *reqB* and *provC*. Component C then *delegates* its required interface *reqC* to component A's required interface *reqA*.

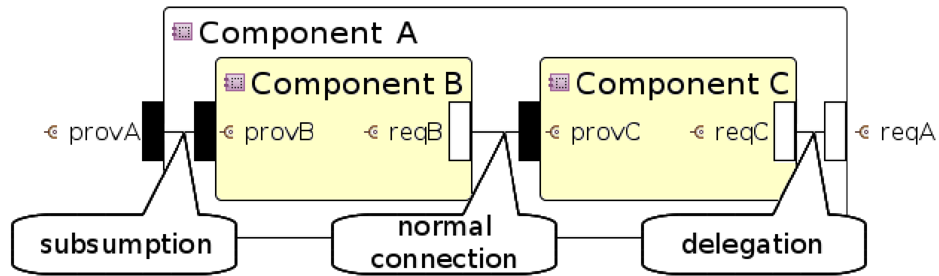


Figure 2.3: Delegation and subsumption

2.2 Connectors

This thesis bases on the view of software connectors described in [6]. Software connectors are seen as first class entities which are used for communication between the components. There are two main points of view on connectors in SOFA 2 - design view and runtime view.

At design time connectors are used to model connections between interfaces of components. It is also at this time that the application designer has to choose the communication style of the connector (e.g. method invocation, messaging,...).

At runtime connectors separate components from the underlying middleware so that the components themselves take no notion of it. This also means that connectors can implement a number of different communication styles which can then be switched only by using different types of connectors.

In the perspective of this thesis (and also of SOFA 2) every interface (no matter whether provided or required) of a component has its own Connector Unit. The Connector Unit then contains a set of so-called Elements which actually implement the inter-component communication.

In this thesis we call server side connectors *Skeletons* while the client side connectors are called *Stubs*.

2.3 SOFA 2

SOFA 2 is a component system mainly developed by the Department of Distributed and Dependable Systems (formerly Distributed Systems Research Group) at the Charles University in Prague. It employs hierarchically composed components which are distributed among a number of so called Deployment Docks running on any computer that is connected to the SOFAnode - a network of computers running the SOFA 2 component system (repository, global connector manager, deployment dock registry and deployment docks).

SOFA 2 is described by [1] and [2], its complete documentation can be found at [4], so the following paragraphs focus only on the parts needed to understand this

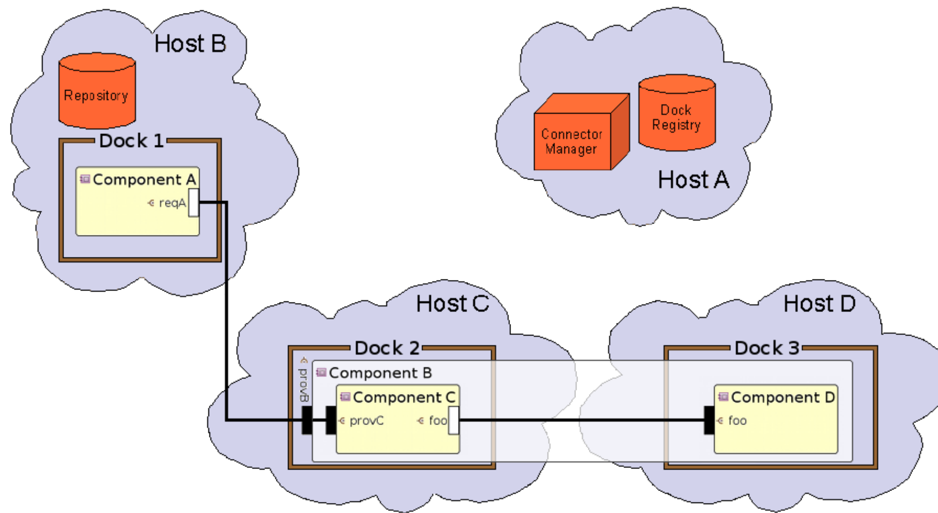


Figure 2.4: Example of a SOFAnode

thesis.

2.3.1 Runtime architecture of SOFA 2

SOFA 2 runtime consists of several separate parts all of which need to be running in order to provide a fully functioning SOFA 2 environment (called *SOFAnode*).

- **Repository**

  The repository is used to store all the information about components in both the form of meta-data describing their structure and relations and the business code of the components (usually both as source code and as binaries stored in jars). It also holds the information needed to deploy components as applications (stored in Deployment Plans) and the libraries and any additional resources the components might need for their execution (in the form of Code Bundles).

- **Global Connector Manager**

  The Global Connector Manager (GCM) is responsible for connecting components together by connecting the parts of connectors of the component's interfaces. In the current implementation it works as a smart RMI registry which also remembers information on the particular connectors.

  Apart from the GCM registry there is also a Local Reference Registry in each Deployment Dock which is used to resolve local objects inside the same Deployment Dock.

- **Deployment Dock Registry**

The Deployment Dock Registry (DDR) is used as a catalog of running Deployment Docks. When a Deployment Dock is started, it connects to the DDR and registers itself within the DDR as soon as possible. The DDR checks the availability of the Deployment Docks it has registered and removes them from the catalog if they are unavailable. The DDR is also the main way of getting references to running Deployment Docks.

- **Deployment Docks**

  Deployment Docks are the actual containers of the SOFA 2 components and applications. A Deployment Dock takes care of the whole lifecycle of the component - beginning with its instantiation, then connecting the component with its surroundings, starting its business code. Deployment Docks look after starting and stopping applications and now newly also after component migration. It does not matter whether the Deployment Docks run on only one computer or if they are spread among a larger number of machines, they always run in separate processes and are thus treated the same as if they were running on different computers.

  Each Deployment Dock has its own name (human-readable) which is unique within the whole SOFAnode and is the Deployment Dock's identifier in the DDR.

  Deployment Docks contain many more meta-structures which are helpful during its work. One of them, especially important, is the Local Reference Registry. Every connector-skeleton of component's provided interfaces is registered both at the GCM and the Local Reference Registry. This concept is described further in the thesis in Chapters 3 and 5.

The Figure 2.4 shows a sample SOFAnode. The SOFAnode contains four computers which are displayed as clouds and named Host A, B, C and D. On Host A there runs the DDR and GCM. Host B hosts the Repository and a Deployment Dock called Dock 1. Hosts C and D contain respectively Deployment Docks 2 and 3. Among the Deployment Docks in this example of a SOFAnode there are instantiated components which were also shown as an example in Figure 2.2. Dock 1 contains component A, component B being a composed component is virtually "spread" across Docks 2 and 3 which contain its subcomponents - components C and D.

Note that a SOFAnode can of course be a lot larger than the example described above, or it can be even launched on a single computer, there are no constraints on the topology of the SOFAnode except that the computers must be available to be accessed by their IP addresses from the rest of the computers of the SOFAnode.

2.3.2 Metamodel

Components developed in SOFA 2 must follow its metamodel. Component's outward appearance is described by its Frame, its internals are described by its Architecture.

The fabrication of components into applications is described by Assemblies and Deployment Plans. Compiled binaries are stored inside Code Bundles. The following paragraphs give a brief introduction into the metamodel of SOFA 2 applications. For a more detailed description see [2].

Frame Component's Frame holds the information about the interfaces the component provides and requires. Interfaces are described by an Interface Type which is just a meta information pointing to the source code containing the actual description of the Interface Type (e.g. a java interface in the implementation of SOFA 2 we are using for this thesis). The Interfaces described by the component's Frame are the only way a component can communicate with other components in the system. This also means that all components which implement the same Frame are interchangeable with each other.

Note that all components have to implement a Frame - even those that have no interfaces (for example the top-level component which represents the application).

```
<?xml version="1.0" encoding="UTF-8"?>
<frame name="frame.Forwarder">
  <provides itf-type="sofatype://iface.Log" name="provLog"
      comm-style="method-invocation"/>
  <requires itf-type="sofatype://iface.Log" name="reqLog"
      comm-style="method-invocation"/>
</frame>
```

Figure 2.5: Example of a component's frame definition

Architecture Component's Architecture describes the internals of the component. It decides whether the component is composite or primitive. In case of a composite component it holds the information about its subcomponents and how they are connected (including the types of the connection - method invocation, messaging, streaming). If the Architecture describes a primitive component, it holds just the component's implementation.

```
<?xml version="1.0" encoding="UTF-8"?>
<architecture name="arch.Logger"
frame="sofatype://frame.Logger" impl="arch.Logger" />
```

Figure 2.6: Example architecture definition.

Code Bundle A Code Bundle is an entity in the repository which contains compiled binaries and possible dependencies on other Code Bundles. For example Code Bundles store the business code of primitive components (their Architectures), the connectors, libraries and so on.

Deployment Plan A Deployment Plan is the final summary of the integration of components in the application. When it is created, the developer has to specify which component is going to be executed on which deployment dock. Then it is *deployed* using Cushion[1] and it is passed to the *Connector generator*. The *Connector generator* then analyzes the connections between components and generates the glue code which is used during the instantiation of components (and now also during their migration) to connect components together. The code of the connectors is then stored inside the Deployment Plan.

```
<?xml version="1.0" encoding="UTF-8"?>
<depl-plan name="deplplan.Logdemo" node="nodeA">
  <depl-subc name="logger" node="nodeA" />
  <depl-subc name="tester" node="nodeB" />
</depl-plan>
```

Figure 2.7: Example deployment plan.

2.3.3 Launching a SOFA 2 application

To launch a SOFA 2 application the first thing needed is a fully functioning SOFAnode with all the necessary parts running (one instance of the GCM, DDR, Repository and all the Deployment Docks needed by the application - described by its Deployment Plan). When an application is launched, the Deployment Docks are actually instantiating components, as described by the Deployment Plan of the application, and connecting them together.

The instantiation starts from the top-level component of the Deployment Plan, which in fact represents the application, and continues recursively through its subcomponents until all components are instantiated. The components are then connected together using the Connector Manager and after that they are started.

The component instantiation part of the deployment process is almost all-round followed in the second stage of the migration process. The differences are described in Section 3.2.4.

[1] Cushion is a command line tool for developing and deploying SOFA 2 applications.

Chapter 3

Implementing migration in a component system

This chapter covers the ups and downs of implementing component migration in a distributed component system. Section 3.1 makes an overall introduction to the theory of migrating components, Section 3.2 covers the implementation itself. The chapter is ended by Section 3.3, which discusses the protocol of transferring the component between two Deployment Docks.

3.1 Migration process in theory

First let us define what migration of components means. In this thesis we understand the migration of a component as taking the component from its original placement and moving it to a new destination. At its new location, the component resumes its work in such a manner, that the application, in which the component is involved, notices as little of the migration as possible.

3.1.1 Primitive components vs. composed ones

When migrating components in distributed hierarchical component systems, it is important to distinguish the basic types of components since their type influences certain parts of the migration process (these differences are further discussed in sections concerning the particular phases of migration). Primitive components are the actual logic of the application they are participating in. They are the only ones which can really be migrated in the sense of moving a process (or rather an object) between different destinations (if they are designed as migrable components - this will be covered further in this chapter).

Depending on how they are implemented by the actual component system, composed components can be implemented in two ways, only one of them actually needing the functionality for supporting migration:

1. As abstract entities which exist only in the model of the application and take no role in the actual code and running processes. This kind of a composed component really is not present in the running application and so there is no need to be troubled concerning this case when designing migration.

2. As a boundary line that has its own connectors really instantiated within the running application and which fully encapsulates its subcomponents since they can interact only with its provided and required interfaces. Migrating these components means reinstantiating their connectors at a different destination and reconnecting them with the outside world (and with the subcomponents of the composed component).

Of course that was just discussing the migration of the actual "instances" of the composed components. Apart from the possibilities mentioned above, migrating composed components could also involve migrating all their subcomponents. However when this possibility is taken into account we implicitly presume that all the components in the system are migrable - and that is not always the case. As is shown in Section 3.2, making a primitive component migrable requires some additional work from its developers and they need not make this decision - they can develop common old-way components which do not support migration.

A question might now rise, why, when we are proposing a way to add migration functionality to a component system, we should retain the possibility of non-migrable components. The answer is simple: because the developers might want to have a component in their application which never changes its place and if that is the case, adding the functionality needed by migrable components would be just a waste of time. An example could be a component using some hardware connected only to a particular computer.

3.1.2 Anticipated vs. unanticipated

When implementing migration of components in a distributed component system, another decision which has to be made is whether the migration is going to be anticipated or unanticipated. The anticipated variant requires the developers of components to add some obligatory functionality to their code which allows the migration to happen. On the contrary, the unanticipated variant should be able to handle everything by itself without the component's code noticing that the migration has occurred.

However, component's internals can always contain some unpredicted functionalities which cannot be effectively handled in a general way. Unanticipated migration of components within the distributed system would be therefore near impossible to accomplish.

Unanticipated migration would require the underlying system to have a very low-level access granting it the possibility to move not only the serialized data of

the component, but also all its threads including their stacks and all the synchronization primitives. Resources like network sockets, handles to files and so on would also have to be resolved and transferred. That would imply too much effort to be taken by the underlying framework.

A way to achieve the possibility of unanticipated migration would be to apply additional heavy constraints on the components and provide a framework which would handle all the resources the components could need (sole usage of the framework would be required on the components). Although the paper does not explicitly state that it describes unanticipated migration, such a method is discussed in [20].

In our prototype implementation we put up with the anticipated variant which requires the developers of the components to implement callback functions in each component which ask it to pause itself, resume, store its state and reload it again.

3.1.3 Overview of the migration process

In the following paragraphs we consider the component only in its general principle and therefore we do not distinguish between primitive components and composite ones - differences in their migration process will be discussed in the appropriate sections which are describing the particular phases of migration.

When migrating a component, several steps, which are more or less independent on each other, need to be performed.

1. **Stopping the component.**

   First the component needs to stop all its jobs and end in a consistent state in which it can be transferred to its new destination. All incoming calls have to be delayed in its connectors only to be forwarded to the new destination of the component when the migration is finished.

2. **Transferring the component to its new destination.**

   When the migrated component is successfully stopped and remains idle, its inner state should be serialized and sent to its new destination where it will be used to recreate the component to its original form.

3. **Reconnecting the component and it's dependencies.**

   In the next step the component practically exists in two separate instances at once. Before the old instance can be discarded, the new one needs to connect its required interfaces to the same components as they were connected before the migration[1] and provided interfaces of the old component need to alert their opposites that the destination of the component has changed.

[1] Why we do not need to handle the possibility that one of those components is migrated before the reconnection will be described in Section 3.2.5.

4. **Resuming the work of the component.**

   In the last step all jobs of the component need to be resumed in a way that the application does not notice that anything has changed (if possible). Since it is quite uneasy to design this step in a general way, we decided to leave the resuming on the developer of the component.

5. **Handling error states during migration.**

   This phase is not actually separated from the others as it is rather orthogonal to them. At any time of the migration process it must be ensured that if anything goes wrong[2], the system can go on in a preferably unmodified state - as if the component was not migrating at all (or as if it has just migrated to the computer it was originally running at).

That was a general description of the migration process. Of course, there are many implementation details, which have to be solved before a first component can be really migrated, but these will be described in Section 3.2.

It should be noted however that the restriction that components can communicate with each other only using the provided and required interfaces solves many problems which could otherwise occur. For example it is not necessary to transport the whole address space, because the component either holds everything it needs within itself or is connected to it using its required interfaces. Of course, there is the problem of other resources like files, network sockets and so on, but the developers of the component must take care of these themselves. Any component which wants to hold a reference to another one can do so only through its required interfaces, which means that there can be no broken references when a component migrates to a new destination.

3.2 Implementing migration

During the work on this thesis a prototype implementation of migration was developed for the SOFA 2 component system. The following paragraphs describe the chosen solutions and show practical examples of the prototype implementation.

3.2.1 Migrable components in SOFA 2

In SOFA 2, components publish their specific features either by implementing interfaces which describe the properties and functionality that the component provides (e.g. the `SOFARunnable` interface which designates that the component has some threads which should be started at the start of the application) or applying annotations on specific parts of the component such as the required interfaces, the class of the component, etc.

[2] For example if someone accidentally kicks off the power supply from the destination computer.

Let us note that implementing the `SOFAMigrable` interface or using the `@Migrable` annotation are the only things needed for the SOFA 2 component system to notice that a component is migrable. Everything else is done inside the Deployment Docks using Reflection API and inside the Connector Units.

The examples in the two following subsections do not show the whole implementation of a SOFA 2 component on a purpose; it would exceed the scope of this thesis. To get more information about writing components in SOFA 2 see [4].

Before continuing to the description of the actual migration process, let us see how a migrable component looks in SOFA 2.

SOFAMigrable interface

In order to specify that a component is migrable, the `SOFAMigrable` interface has been introduced.

```
public class MigrableComponent implements SOFAMigrable, ... {
  ...

  public void pause() {...}
  public void resume() {...}

  public void saveState(Map<String, Serializable> storage)
    {...}
  public void loadState(Map<String, Serializable> storage)
    {...}

  ...
}
```

Figure 3.1: A migrating component in SOFA 2 using the SOFAMigrable interface

Figure 3.1 shows one way of adding the migration functionality to a component - implementing the `SOFAMigrable` interface. The interface defines four methods that have to be implement by the developer of the component to support seamless migration.

Let's observe the methods in more detail:

- `void pause()`

  This method is probable the most important one needed by migration. As its title indicates, this method should pause all the internal jobs of the component. It is presumed that when the method has returned, the component makes no more changes to itself. The `pause()` method should stop all the threads which can be running inside the component and make sure that there will be no more

outgoing calls from the component (they are stopped inside the connectors nonetheless).

Correct implementation of this method is crucial for the migration to go on without errors.

- `void resume()`

  The `resume()` method is, literally speaking, an inversion of the `pause()` method. It is executed as the last part of the migration process when the component is both instantiated and connected with the rest of the application.

  The `resume()` method should take the component (with all data on its place) and return it to its normal functioning - i.e. start again all the threads, allow calls if they were disallowed by the `pause()` method and so on. No further assumptions about the `resume()` method are made by the framework than that it starts again the component when it was previously paused.

- `void saveState(Map<String, Serializable> storage)`

  To transport the state of the component to its new destination we decided not to use the serialization mechanism[3]. Instead the developer of the component is required to implement method for saving the state of the component. This allows the developers to have more control of the process. The class of the component thus does not need to be serializable (and also its members do not have to be serializable) and can transfer more information this way than with serialization - for instance it can transfer files it uses.

  The `saveState(...)` function is executed on the component after it was successfully paused.

- `void loadState(Map<String, Serializable> storage)`

  Just as the `resume()` method is an inverse for the `pause()` method, the `loadState(...)` method is an inversion of the `saveState(...)` method. It is invoked on the component when it is instantiated at its new destination after it is connected to the rest of the application but before the `resume()` method is called to resume the component's jobs.

  This method should restore all the resources the component is using to their original state.

Using annotations to mark a migrable component

The second option the developer has to designate that a component is migrable, is to annotate the component's class with the `Migrable` annotation.

Figure 3.2 shows a migrable component implemented using the annotations. The existing annotations of the SOFA 2 system were extended by:

[3] This choice is discussed in Section 3.2.3.

```
@Migrable
public class AnnotatedMigrableComponent {

  @PersistentAttribute
  Serializable someData;

  @Resume
  public void resume() { ... }

  @Pause
  public void pause() { ... }

  ...
}
```

Figure 3.2: A migrating component in SOFA 2 using annotations

- `@Migrable`

  When using annotations for the implementation of a migrable component, the `@Migrable` annotation is mandatory in order to tell the underlying functionality that the component is migrable. This is the only annotation concerning migration that a migrable component cannot do without - the other annotations are just optional.

- `@PersistentAttribute`

  Instead of having separate functions for saving and loading the state of a component, the developer of a migrable component can choose to just annotate the actual fields they want to be persistent during migration. The constraint is, of course, that the fields have to be serializable. Values of fields not annotated with the `@PersistentAttribute` annotation are not transferred to the new location of the component and have default values.

- `@Pause`

  The `@Pause` annotation is used on the method which has to be run in order to pause the component's jobs. It is the same as if it were the `void pause()` method declared in the `SOFAMigrable` interface.

- `@Resume`

  Just as the `@Pause` annotation on a method replaces the implementation of the `pause()` method of the `SOFAMigrable` interface, the `@Resume` annotation on a method replaces the implementation of the `void resume()` method. Methods

annotated with the `@Resume` annotation are called after the instantiation of the component at its new location to resume the component's jobs.

It is possible to use both the `SOFAMigrable` interface and the annotations. It does not bring about any additional possibilities though.

3.2.2 Stopping the component's jobs

The first step of the component migration is stopping the component so that it can be easily transferred to its new location where its work is then resumed.

In [7] there is stated that:

> ...a component can be consistently reconfigured only when the following conditions are fulfilled:
>
> - Its clients carry out no new invocations on it.
> - The invocations of its clients on it have been completed.
> - It carries out no new invocations on any other components.
> - Its invocations on its server components have been answered.
>
> When a component fulfills the above conditions, we say, the component reaches a reconfigurable state.

In short, a component is in a reconfigurable state when it currently does nothing and no new calls are passed to it.

Satisfying the precondition "*Component's clients carry out no new invocations on it.*" is simple at the first glance. Since all communication between components is carried out through the connectors of their interfaces, all that needs to be done is notifying the provided interface connectors that they should not pass further any new invocations on the component.

In [7] the connector is represented by a *Virtual Stub*. However, they only have one connector for the whole connection that is on the client side. This means that all the stubs need to be informed to delay their invocations. On the contrary, since in SOFA 2 there are both client side and server side connectors, we delay the incoming calls in *skeletons* - on the server side. This allows us to generate less communication during the process and also less utilize the knowledge of the structure of a deployed application (which is available in SOFA 2 but it may not be available in other component systems).

To provide the delay of incoming calls, the server side connectors were extended with methods `pause()` and `resume()` which notify the connector either that no new invocations should be passed to the component or to continue the execution.

However, delaying incoming calls brings about another problem: in case of synchronous communication, if all the incoming calls are delayed, there can emerge a dead-lock caused by cyclic dependencies. For example suppose that we have two

```
public void serverMethod(...) {
  while(paused)
    wait();
  ... // other connector logic
  // call the method of the component
  component.serverMethod(...);
  ... // other connector logic
}
```

Figure 3.3: Example of code used to delay calls on a method called "serverMethod".

components, A and B, both have some provided and some required interfaces and are interconnected. We want to migrate component A to a new location so we delay all incoming calls on component A. However, component A has just called a method on component B which results in component B calling back component A - and the method called first from component A cannot be finished because the component B's subsequent call is delayed - a dead-lock.

Figure 3.4: Example of a possible dead-lock

In order to prevent the dead-lock, incoming calls must be divided into those which could cause a dead-lock and those which would not. The "safe" ones can be securely delayed while the calls which would cause a dead-lock in case of their delay have to be let in.

In [7] they come with a solution based on resolving so called "dynamic dependencies" - a dynamic dependency arises between a client and server on a method when the client executes the method on the server. It is removed as soon as the method returns back. The detection of dead-locks then consists of traversing so called "call paths" which are formed from the dynamic dependencies. If a call path contains a cycle, there would be a dead-lock and the call must not be delayed.

In the prototype implementation, the stopping mechanism was partially inspired with the above solution, but it took the advantage of the fact that SOFA 2 always passes a *Call Context* along with the calls. The *Call Context* is used to uniquely identify threads within the SOFAnode in such a manner that it can be easily discovered if two threads originated from the same place or even if one thread is an ancestor of another. To take advantage of the *Call Contexts* to prevent dead-locks

during the pausing of the component and delaying its incoming requests, an auxiliary structure called *Thread Observer* was introduced. Each component has its own *Thread Observer* that is notified from the provided and required interfaces about all the threads which concern the component - either they enter the component's provided interfaces or leave it through its required interfaces. The *Thread Observer* remembers all the threads within the SOFAnode that the component participates in. If it is found that the component already participates in a thread which tries to enter it through its provided interfaces, it means that there would be a dead-lock if it was not let in.

To ensure that all the on-going invocations (both incoming and outgoing) of the migrated component are finished, the *Thread Observers* are used to monitor the number of operations that the component's connectors are waiting for. When the number of threads the *Thread Observer* monitors is at zero, the component has no ongoing calls and according to the above mentioned preconditions is in a reconfigurable state and therefore it can be migrated.

The solution that is proposed in [7] however has a little flaw - it does not count with the possibility that a component can be changing its inner state itself (for example when it has multiple threads running which do some nontrivial logic and store their results within the state of the component). This problem would be near impossible to figure out on the general level of the component system. For its solution the ostrich method was chosen - the developers are asked to solve the problem for the underlying system by implementing the `pause()` method on their components.

It is silently presumed that the `pause()` method (and indeed also all the other methods of the `SOFAMigrable` interface) is implemented correctly and that after it has finished, there will be no more changes to the component from its inner code and that the component will invoke no more outgoing calls. But the outgoing calls that the component might invoke are still discarded inside the appropriate *stubs*, just as a precaution.

3.2.3 Saving the state of a component

There are generally three approaches of handling the component's state during its migration.

1. **Implicit serialization**

   Serialization is the first thing that comes in mind when it comes to saving the state of a software component. It is available in most modern languages and systems, it is easy to use, and requires almost no additional code from the developer. It was not chosen though to save the state of the component because of some of its undesirable consequences (e.g. all the contains of a serializable component need to be also serializable) and lack of control over the process.

2. **Explicit serialization**

Another option is to pass the process to the developers. They can then decide which approach suits them best. Thanks to this alternative the developers of components also have many more means of control over the process of transferring the state of the component to the new destination.

3. **"Stateless" components**

   The last possibility is for the components not to have control over their inner states at all. All the data and resources could be managed by the component framework and so the framework would have absolute knowledge about the component's internals. This approach requires the system to provide a large and sophisticated framework for managing the resources components might want to use. Since SOFA 2 never counted with this possibility (which is also rather restrictive), we did not use this possibility of saving the component's state either.

   This approach is described for example in [20].

In the prototype solution the second option was chosen, passing the responsibility of saving the component's state to its developers. Each component has to implement the function

```
void saveState(Map<String, Serializable> storage)
```

which is called from the outside to store the component's internals in the object passed as a parameter (`Map<String, Serializable> storage` is in fact just a HashMap supposed to contain name-value pairs) or annotate the appropriate fields with the `@PersistentAttribute` annotation.

Handling resources used by the migrated component

A component can be using many resources which are not related to the underlying component system - e.g. handles to files, network sockets, database connections and so on. In this case it is more than useful to utilize our approach. The developer can use the storage object even to transfer files (if the expected network connection is fast enough).

The Serialization option would bring huge difficulties when coping with additional resources of the components and would probably result in requiring the developers to implement some additional functionality nonetheless.

It is clear, that the method most suitable for handling the component's resources during its migration is the third one - requiring the developers of components to use purely the resource management framework provided with the component system.

3.2.4 Re-instantiating the component

The process of reinstantiating the component is probably the most straightforward part of migrating a component. It is also highly dependent on the implementation of the component system.

The reinstantiation process generally follows the process of normal component instantiation. The main difference is that the newly instantiated component should begin in a still state in which no process is being executed on it. It is also necessary for our solution to instantiate the component unconnected to other components. This is caused by the SOFA 2 component system where components are always instantiated according to the Deployment Plan of their application. However, since the Deployment Plan is static and was created before the application was launched and it does not change during the runtime of the application, it can have no knowledge about the actual layout of components among the Deployment Docks.

To compensate for the difference between the Deployment Plan of the application and its actual layout changed by migration, we had to come up with a way of passing the references of the components during migration. This will be discussed further in Section 3.2.5.

Apart from instantiating the component unconnected to its opposites, the component also needs to be prepared to load back its state before its jobs can be resumed. That is why it was declared that the component needs to be in a still state in which there are no executed processes within the component: to assure that it will not change its state until it has been allowed to resume its work at its new destination.

3.2.5 Reconnecting the components

When the component has been successfully reinstantiated at its new destination, the next step is reconnecting it with the other components. Since there can be several different communication styles implemented in the component system, the process of reconnecting the interfaces has to be generalized as much as possible in order to enclose the migration intelligence concerning the actual communication styles only in the connectors that implement it. If every communication style required to have its specific migration code present in the runtime of the component system[4], it would lead into code explosion and lack of lucidity. Every new implementation of a communication style would also have to watch out if it does not badly influence the migration of other communication styles.

In the prototype implementation for SOFA 2 this was done by introducing a common interface for migrable connector elements. The interface separates the migration of the connectors into three callback functions:

- onBeforeMigration(Map<String, Serializable> parameters)
- onNewInstanceCreation(Map<String, Serializable> parameters)

[4] In SOFA 2 this means in the code of the Deployment Docks.

- `onAfterMigration(Map<String, Serializable> parameters)`

With the `pause()` and `resume()` methods of the server-side connectors, only these three methods need to be implemented in the connectors to implement migration for a newly introduced communication style. Using this concept, it should be unnecessary to make any other changes to the system when a newly added communication style is to be used with migration.

The `parameters` of the methods are used as in/out means of passing data between the subsequent calls of these functions on the old instance and the newly instantiated one.

The `onBeforeMigration` method is invoked on the old instance of the connector to tell it to prepare for migration and get the data necessary for the new instance to be able to work correctly(for example the reference to the target of a required interface).

At the instantiation at the new location, the new connectors are prepared by calling the `onNewInstanceCreation` method. The changes made to the `parameters` in the `onBeforeMigration` method are of course passed to this method also.

After the instantiation of the component at its new location, the `onAfterMigration` method is invoked on the old instance of the connector.

Migration with method invocation communication

The migration of method invocation connectors can be divided into two parts, that can be handled separately and which require different approaches - 1. reconnecting required interfaces and 2. reconnecting provided interfaces.

The correctness of the process described in the following paragraphs is discussed in Section 3.3.

Required interfaces

Even though there are still a few issues that need to be solved while reconnecting the required interfaces, this step is a little bit more straightforward than the reconnection of provided interfaces.

One of the problems associated with reconnecting required interfaces of a component is that the system has to adapt to possible modifications of the application structure which could have been performed during the application's runtime (i.e. previous component migrations). To provide the reinstantiated component with up to date information about the location of the components the required interfaces are connected to, the old instance of the component must pass the *Remote Reference Bundles* of its required interfaces to the new location. There they are used to connect the new component to its opposites.

Provided interfaces

One of the big differences when reconnecting the provided interfaces is that while most of the code concerning the reconnection of required interfaces takes place at the new destination of the component, the reconnecting of provided interfaces happens mainly at the old location. There are also more complex issues to be solved - mainly in the actual migration protocol and communication with the opposite components.

In the prototype implementation, the provided interfaces are not reconnected at the same time as required interfaces. To enable easier handling of possible errors, provided interfaces are reconnected just before the old instance of the component is discarded. This decision is further described in Section 3.2.8.

When designing the reconnection of provided interfaces during the work on the thesis, the first option that was considered was to employ the knowledge about the assembly of components. Components which are connected to the provided interfaces of the migrated component would then be traversed and manually asked to rebind their references. This choice was not implemented though for its dependence on the knowledge of the application structure, not to mention the need of distinguishing primitive components from composed ones and adapting the code to each possibility. Instead a method based on passing specialized remote exceptions, that does not require the knowledge of the application structure, was developed.

Just as the reconnection of required interfaces needs the *Remote Reference Bundles* from the old component instance, the reconnection of provided interfaces needs to be passed information about the newly instantiated component. The process of reconnecting the provided interfaces takes place in the provided interfaces of the old component and in the required interfaces of the components which are connected to them (or in the provided interfaces of the parental components which are subsumed to the migrated component).

The provided interfaces of the old instance of the component are informed that the component was migrated to a new location and the *Remote Reference Bundles*[5] describing the new location of the appropriate provided interfaces are passed to the old skeletons. When a request enters a skeleton of the old instance of the component, the skeleton throws a `MigratedException` to which it passes the *Remote Reference Bundle* of the new skeleton. A similar solution was described for example in [15].

The `MigratedException` inherits from `RemoteException` and is used internally by the connectors for passing information about the migrated provided interfaces and telling the required interfaces that they should reconnect their references. When the `MigratedException` is caught at a stub (i.e. at the required interface) the client gets the *Remote Reference Bundle* from the exception and reconnects its references appropriately.

[5] I.e. the local references and remote references of the skeletons - this will be described in the following section.

```
void call(Object [] args) {
  ...
  try {
    // try the call
    target.call(args);
  } catch(MigratedException exc) {
    // if the component is migrated
    // rebind the reference
    rebindTarget(exc.getRemoteRefBundle());

    // and try the call again
    call(args);
  }
  ...
}
```

Figure 3.5: Example of the MigratedException usage

Another way of taking care of the provided interfaces (also described in [15]) would be to leave the references as they are and transform the *Skeletons* of the old instance's provided interfaces into *Forwarders*. These *Forwarders* would have a reference to the new instance of the component and forward the calls to it. However, this attitude is not very advisable. Each time a component with a provided interface would migrate, the chain of *Forwarders* would be longer and longer. That would produce too much overhead and so this option was discarded in the prototype implementation of this thesis.

Deciding whether to use local or remote communication

Orthogonal to the problems with the reconnection itself is the decision whether the destination is in the same address space or not - i.e. whether to use local or remote calls. This is not as much of a problem as it is more an optimization since remote calls can be used any time. But remote calls tend to considerably slow down the system so it is favorable to be able to distinguish the cases when local calls can be used instead of remote ones.

The solution was fairly simple to implement in the SOFA 2 component system. Every Deployment Dock also comes with a Local Reference Registry[6]. Since there can be only one Deployment Dock running within one address space (meaning in one process), every address space also has its own single instance of the Local Reference Registry[7] which contains references to all the components' skeletons and their stringi-

[6]Can be seen as RMI registry, but operates only in one address space.

[7]In SOFA 2, the LocalReferenceRegistry is implemented as a pure static class.

fied references. Knowing that the names of Deployment Docks are unique within the SOFAnode it was possible to alter the process of stringifying the references in the Local Reference Registry so that the stringified references also contain the name of the Deployment Dock they reside in. This made every stringified reference also unique within the whole SOFAnode.

Information needed to get the references to the server side connectors are passed within SOFA 2 using so called *Remote Reference Bundles*. Formerly the *Remote Reference Bundles* contained only one reference (local or remote) which directly specified the type of connection. Also if the *Remote Reference Bundle* contained only a local reference, it was impossible to pass them to components residing within other Deployment Docks - they would be useless because of the purely local character of the local stringified references. To make it available to decide whether to use local or remote connections the *Remote Reference Bundle* now contains both stringified references - local and remote.

Thanks to the local references being now distinguished by the Deployment Dock their targets are dwelling in, their stringified references have a relevant meaning on any Deployment Dock of the same SOFAnode. When the reference is being resolved, the local reference is tried first and if it fails, the remote reference comes in its place. In Figure 3.6 there is shown an example of the reference binding code.

```
// try the local reference
localRef = remoteRefBundle.getRef("local");
target = LocalReferenceRegistry.unstringify
          (localRef.stringifiedRef);

if(target == null) {
    // the local reference failed
    // use remote reference instead
    rmiRef = remoteRefBundle.getRef("rmi");

    java.rmi.registry.Registry rmiRegistry = getRegistry();
    target = rmiRegistry.lookup(rmiRef.stringifiedRef);
}
```

Figure 3.6: Sample code deciding whether to use local or remote calls.

Migration with messaging communication

The second communication style for which migration was implemented during the work on this thesis is messaging. SOFA 2 uses JMS [11] as middleware for messaging communication, therefore the following paragraphs will be using JMS terminology.

Without limiting the generality, reconnecting messaging connectors can focus only on 1:n communication using the JMS topics[8].

Just as with the reconnection of method invocation connectors, the reconnection of messaging connectors can be distinguished between the reconnection of required and provided interfaces. However, when migrating the messaging connectors, the broker service also has to be taken into account. Since the broker can be running "almost anywhere", let us focus on it first.

The processes taking place when reconnecting connectors for messaging can take advantage of the fact that any connector can both accept messages from the topic and send its own. Should the component running the broker be migrated, a new instance of the broker could be created and then the information about it could be sent as a special message in the old one. The connectors would then just use the new broker. The only thing needed is to delay the outgoing messages until every connector is rebound to the new broker.

When reconnecting required interfaces with the messaging communication style, the connectors are rebound absolutely the same way as when reconnecting method invocation required interfaces. They pass the `RemoteRefBundle` to the new destination where it is used to bind to the JMS topic again. The difference between messaging and method invocation required interfaces is, that they do not have to distinguish between local and remote calls - the messaging service is used always the same way.

The reconnection of provided interfaces is a bit different both from required interfaces in messaging and provided interfaces with method invocation communication style. Messaging communication is usually asynchronous (and without limiting the generality, the solution can focus only on the asynchronous case) and on top of that, a required interface can be connected to several provided interfaces. Therefore, even though message sending could be delayed until a component connected with the provided interface would be migrated, this case would slow down all the other components as well.

To prevent such a slow down of the system during migration, when migrating a messaging provided interface, the connector is set to store all incoming messages that occur during the migration. Every messaging connector was also extended with the knowledge of a unique id that it possesses. When the new instance is created and connected to the topic just the same way as required connectors are, it sends its id to the old one. The old instance of the provided connector then uses the passed id of the new instance to send forward all the messages that it delayed during the migration to the new instance.

The new instance of the provided connector has to be started in a delaying mode just as well. To prevent the old instance from resending messages which the new instance could already have received, it stores all the messages it receives since its

[8] Communication using a message queue can be handled as communication with a topic with only one recipient.

instantiation in a queue. When it receives a message resent by the old instance that it already has in its queue, it notifies the old instance of the connector to stop receiving and resending the messages. It simply processes all the messages in its own queue and when the queue is empty, resumes working in a normal mode.

Other communication styles

In the above sections the text focused mainly on method invocation and messaging communication styles. This is because they represent the two most wide-spread types of middleware. These are also the two styles of communication for which migration was implemented in the SOFA 2 component system during the work on the thesis. However, the main principles stand still even for other communication styles and a rule of a thumb this thesis is trying to follow is, that the reconnection of interfaces during the migration of a component should be mainly the concern of the connectors themselves, not the containers of the components (Deployment Docks in case of SOFA 2).

This attitude was successfully tested also on the prototype implementation developed for the thesis. The Deployment Docks contain only general code invoking the callback methods of the connector elements. Both the method invocation connectors and the messaging connectors then take care of their migration within their own code. Their migration required no changes on the migration functionality of the Deployment Dock.

3.2.6 State loading and resuming component's work

At this stage the component should be successfully reinstantiated and connected to its surroundings. To finish making it an effectual replacement of the original instance and thus bringing the migration process to a close, it only needs to load its previous state and resume its jobs. In the prototype implementation this is handled by the `loadState(...)` and `resume()` methods.

The state of the component needs to be loaded first. The process is of course dependent on the method chosen for saving the component's state, but it will most probably need some code to be written by the developer. They should help loading back the resources which would otherwise be hard to manage purely on the general level of the framework - e.g. database connections, network connections, file handles, specific hardware initialization and so on (this can be of course done also in the resuming part, but that would break the logical separation of the two methods).

When the component has its state loaded, it can resume its jobs. This in fact means it can start working inside and generating requests on other components. In this stage the component still will not be able to receive any requests - all other components still have references only to the provided interfaces of the old instance of the component. The references need to be redirected to the new location, but this is not done till the cleanup part described in Section 3.2.7.

3.2.7 Finalization

The migration process is almost over now. The component is sitting at its new place, happily sending requests to other components and if there occurred no error during the migration, the old instance of the component can be discarded. Before that is done, the interfaces have to be informed that the component was migrated to a new place and to adapt accordingly. In case of method invocation provided interfaces, this means to send `RemoteReferenceBundle` using the `MigratedException` as described in the subsection concerning the reconnection of provided interfaces.

Discarding the component can vary depending on whether the programming language which was used to implement the system has a garbage collector. Either way, the best thing to do is use the system's component stopping functionality (if there is one) and drop all the known references of the actual content of the component to allow possible garbage collection. Outgoing calls are already stopped since the stopping phase, so the component should have no means of communication with the outside world in case its developers did not follow the recommendations when implementing the `pause()` method of the component.

3.2.8 Handling error states during migration - reverting back to the original component

The migration process is insidious in that if something goes wrong during the migration (e.g. the computer with the Deployment Dock which should have been the new location of the component is no longer available), the system has to be able to continue its work in an unmodified state. To ensure that the system can revert back from any point during the process of migration, no irreversible changes can be done to the old component until the new instance is in place and fully functioning. This is also why the reconnection of provided interfaces was left after all the other steps - to be certain if there occurs an error during the migration, that the surrounding components still have a reference of the working component.

If the old instance of the component is left in a reversible state until it is clear that the new instance can take over its duties, it can be easily reverted back to its functioning any time an exception is detected during migration. Whenever there occurs an error during migration, the new instance can be simply discarded and since the old instance was only paused, it can resume its jobs as if nothing has happened.

It should also be noted that the `pause()` and `resume()` methods of the migrable components are presumed to be implemented in such a way, that if they are called on the same object one after another, the object normally continues its functions as if almost nothing happened.

3.2.9 Possible problems with concurrency

There might occur some problems if the component is tried to be migrated several times at once (from different threads or processes). Since the migration process is quite delicate and should happen preferably as an atomic operation, concurrent migration of the same component has to be precluded. If a component would be tried to be migrated by several threads at once and the system would allow it to happen, it could result in the component being instantiated on several computers at once. Not to mention that other components could be given references to any of those instances.

3.3 Handover protocol

In the above sections there are described the principles of migrating a component and it is discussed how to implement the migration process so that it is as transparent to the application and as seamless as possible. The following paragraphs focus on summing up the hand-over protocol when reconnecting the components and proving that the reconnection is logically consistent and cannot break causal dependencies of the calls that might occur during the migration. There is also discussed what happens when two or more reciprocally dependent components are trying to be migrated at once.

3.3.1 Migration influence on the rest of the application

When discussing the correctness of component's migration, the influence on the rest of the application should be thought over first. According to common sense, the only component (if any at all) of a running application which should notice the migration, is the migrated component itself.

The proposed solution manipulates only with the internals of the migrated component. The rest of the migration process takes place in Deployment Docks and the connectors of the provided and required interfaces. Hence the only part of the components remaining in place which notices the migration, lies in their interfaces. Since the interfaces are not a part of the component's code, no component that is not migrated can notice that migration takes place.

3.3.2 Method invocation - incoming calls during migration

During the migration of a component, requests which are sent to it have to be delayed in the connectors. While reconnecting the method invocation provided interfaces, first the request is rejected with the `MigratedException` informing the required interface of the opposite component to reconnect to the new instance. These steps could break the causal dependencies of the incoming calls so the contrary has to be proved.

Demonstration of correctness

To prove that the migration handles correctly the requests which are generated on the migrated component during its migration, the rules that have to be obeyed in order to call the communication correct have to be defined:

1. Any request laid on the component is processed no matter the odds.
2. Any two requests invoked in a synchronized manner are processed in the same order as they were called.

The requests mentioned in the first constraint can be divided into those before the migration starts and those after the migration begins. It is clear that all requests which arrive before the migration starts, are answered according to the common ways of the component system. They either finish and are returned back before the migration starts or they are processed during the first stage of migration when the component is being stopped. Just to remind of the stopping mechanism - when the component is being stopped, all its running jobs are finished before the process passes to the next stage.

Requests which are received during the migration are delayed inside the *Skeletons* of the migrated component until it is successfully reinstantiated at its new location. If the component cannot be moved to the new Deployment Dock or if any error occurs during the migration, the old version of the component is told to continue its jobs and the delayed requests are passed to the original component. When the component is seamlessly transferred to its new destination, the delayed requests are passed further and treated just like any other requests generated after the migration. They are returned back with the `MigratedException` to the *Stubs* of the components that generated them. Inside the *Stubs* the exceptions are parsed, the *Stubs* are reconnected to the new destination and the requests are sent again.

The above two paragraphs prove the first point. As for the second one, it is obvious that if the two requests come from the same thread, the condition is fulfilled. Since the requests are synchronous and therefore the code waits for a request to finish before it continues further, no request can be invoked until the previous one is done.

The rest of the incoming requests comes either from different threads of the same component or different threads of different components. In case they were invoked from two different threads of the same component, they were either synchronized using synchronization primitives or nothing can be said about their synchrony. If they were synchronized using synchronization primitives, their invocations were serialized and are the same as if they were invoked from the same thread.

If the requests are invoked from separate components, their synchronization would have to be either implemented by the components themselves or by the underlying component system. If the developers of the components decide to synchronize the calls on their own, the reconnection during migration shows off just as a delay caused by the network and middleware. In this case the migration process knows

nothing about the synchronization but also does not cripple it in any way. If the component system supports synchronization amongst components, it should also be reflected in the connectors somehow. However, in our prototype implementation we did not have to solve such problems because SOFA 2 does not support synchronization of threads amongst different components. The only way that two calls from threads from different components could be synchronized would be that they emerged from two threads which were synchronized among one component and then traversed other components to end up in the migrated one. In this case the synchronization is actually the same as synchronization in one component, which we have shown makes no problem.

3.3.3 Reconnection of messaging connectors

The correctness of migration of messaging components is quite different to define. Messaging communication in SOFA 2 is asynchronous and causal dependencies are not enforced. Therefore the correctness of component migration when messaging is concerned should focus mainly on three conditions:

1. **There occurs no loss of messages during migration.**

   If the messaging middleware used to implement the messaging communication in the component system does not lose any messages, there should occur no message loss when a component is migrated. It is easy to see that the proposed migration procedure fulfills this condition. When the component is stopped during migration, it stores all the incoming messages in a queue. As the new instance is prepared for functioning, all the messages stored in the queue of the old instance are resent to the new one which then processes them. This way no message should be lost if the messaging middleware is reliable.

2. **No message is processed more than once.**

   If a message should be processed more than once, it would mean it is processed either both at the old instance of the component and the new one, or twice at the new instance. For the message to be processed more than once at the old instance could occur only if it were received more than once and this should be omitted by the underlying messaging middleware. The message cannot be processed at both the old and the new instance of the component. If the old instance processes the message, it means it was not stopped and so the message is not queued to be resent to the new instance.

   A message could be received more than once (actually twice) by the new instance of the component in only one case. The message would have to be sent after the new instance was already running and also resent by the connectors of the old instance. However, since the messages at the new instance's connectors are stored in a queue until it receives all the messages generated during

migration, it is more than easy to check the messages for duplicity. For this check to be possible, each message has a unique id.

3. **The messages are processed in a non-changed order.**

   Presuming that the underlying middleware preserves message order when the communication occurs between two components, this condition just means that the old instance's connectors should resend the messages accepted during migration in the same order as they were received. The new component instance's connectors should then store all incoming normal messages until all the resent messages are received and then continue first from its message queue.

3.3.4 Migration of a larger number of components

When a number of components is migrated in the system, several notable situations might arise. The first of them is, that a component migrates several times in a row and other components do not rebind their required interface connectors in time. That is, before the component is migrated again. In this case the consistency of the application holds. There will only be some overhead as the required interfaces trace the subsequent locations of the migrated component until they get a reference to a live instance. The same goes for the case when a component is migrated before it can rebind its interfaces to new locations of components that were migrated before itself.

Concurrent migration

A problem might arise when the system migrates several components at once. Let us presume that there are components A and B which both have provided and required interfaces connected to each other, thus forming cyclic dependencies. If component A is asked to be migrated just after it sends a request to component B and component B sends a request to component A and is migrated before the request from component A arrives, there can be a deadlock. In [7] this problem is solved by keeping track of the so called *dynamic dependencies* and searching for cycles in the graph. The overhead generated by keeping the graph of *dynamic dependencies* however seems quite large to be neglected.

Instead of tracking the *dynamic dependencies*, the prototype implementation simply uses a lock at the level of the whole *SOFANode* that is set whenever a component is migrating. Thus even if several components are trying to migrate, the migration takes place only once at a time.

3.3.5 Possible problems of the method

The only major issue that we can see in the proposed protocol is that the Deployment Dock which contained the old instance of the migrated component has to be running until all the provided interfaces are reconnected - i.e. until all the components which were connected to them try to invoke a request. If the Deployment Dock is shut down before all the interfaces are reconnected, when the components invoke a request on the migrated component, the system will crash. This is caused by the fact that the *Skeletons*, which the opposite component's *Stub's* references were pointing to, will no longer be in their places. The system has no means of restoration in that case.

This is a serious problem indeed. It was not solved in the prototype implementation because it is presumed that the Deployment Docks are running faultlessly at their place and are shut down only if the whole SOFAnode is quit. Should this problem be untwisted, the easiest solution would be to extend the *Stubs* and *Skeletons* with a dummy method used for checking whether the opposite has migrated (let's call the method `checkConnection`). The method would contain no code except for the functionality which handles migration of components.

The `checkConnection` method could then be periodically executed to check whether the referenced component is still in the same place (but this solution would generate additional unwanted network traffic). Another possibility would be to call the method only when a component migrates. For optimization purposes it would be good to distinguish the components which are dependent on the migrated component but it is not necessary. Since the method would be executed only once for each required connector per migration and migration needs some nontrivial time to finish nonetheless, the system would do even if all the components executed the `checkConnection` method at once.

3.4 Improvements

The general migration procedure described above, if it is implemented blindly and just following the main steps, has one notable flaw. Since the component is paused during the very first step of the migration algorithm, the system has to wait all the time until the new instance is instantiated and set up to substitute the old instance. This means that it counts on the idea of component instantiation being so fast that it does not slow down the migration process.

However, if components need some nontrivial functionality to be instantiated (for example network communication with a repository as in SOFA 2), the instantiation of the component at its new location can take quite a significant time. When the algorithm was first implemented for the SOFA 2 component system and some

measurements were made, the reality was, that for components with the inner state size of less than about 1 MB, the component instantiation and preparation took most of the migration time. Even a stateless component took hundreds of milliseconds to be migrated and pausing the component for such a long time was undesirable.

When the above proposed migration procedure is viewed with the instantiation duration problem in mind, it is easy to see that the component instantiation at the new location is independent from the component's inner state and its processes. The new instance of the component at its migration destination can therefore be created before the component is paused. This is even based on the idea of using fault tolerance to implement migration - to have a secondary instance of the component prepared at a different `DeploymentDock` and synchronize the inner state of the component with the second instance. When the component would want to migrate, all that would be necessary to do would be to reconnect the interfaces.

If fault tolerance is not present at the component system, the idea can be implemented as follows. The reinstantiation phase of the migration is split into two - preinstantiation and state loading. Stopping the component can then be moved after the preinstantiation phase. The migration process then looks:

1. **Preinstantiation**

   The preinstantiation phase utilizes the fact, that loading a new instance of a component into a `DeploymentDock` does not influence the running system at all. The first step of the migration process can thus be instantiating a clean component at the migration destination. Of course, it has to be instantiated into an inert state in which it does not communicate with the rest of the system, but this is the same case as in normal migration.

   During this phase, even the required connectors of the migrated component can be reconnected. This choice is only optional since reconnection phase is due to come some time later, but it can lower the amount of communication needed in the phases when the component is stopped - these are desirable to be as fast as possible.

2. **Stopping the component**

   The phase in which the component is stopped and all the communication sent to it is delayed, is moved to the second place. This way, only the phases that need the component to be stopped really lengthen its inaccessibility time.

3. **State transfer**

   When the new instance of the component is prepared at the migration destination and the old one is successfully stopped, the state of the component has to be transferred to the new instance.This step is the first one which really needs to component to be stopped.

   If the inner state of the component is very large and does not change too thoroughly and too often, the state transfer phase could be also split into

preparation and finalization. The preparation phase could transfer the state of the component without the need of stopping it. During finalization, when the component would be stopped, only the difference between the state transferred by preparation and the state which the component would really have could be sent. This might reduce the time needed for migration of components with a large inner state.

4. **Reconnecting, Resuming, Finalization**

   The reconnecting, resuming and finalization phases remain the same as in the migration process described earlier. Only the reconnection focuses just on provided interfaces, because required interfaces are reconnected at the preinstantiation phase.

The advantage of this improvement is that it does not need any special functionality to be added when component migration is implemented in the component system the way described in the previous sections. It only "shuffles" a bit the code concerning migration and requires it to be well structured. The differences between using the unmodified migration procedure and the improved one are described in Chapter 6.

3.4.1 Preinstantiation and concurrency

There is one more reason why the modification of the migration process by introducing preinstantiation can be beneficial. The reason is, that since the preinstantiation phase does not influence the rest of the system, it enhances the possibility of paralleling the migration of a number of components.

If there should occur a migration of several components, the preinstantiation phase of their migration can be taken concurrently. In an ideal case this would mean that migration of several components would take only as much time as preinstantiating "one" component (since all the preinstantiations would be run simultaneously) and then the times needed for serial finishing of migration of the components. As mentioned in Section 3.3.4, the infrastructure needed for completely concurrent migration of components searching for and respecting dynamic dependencies between the components would produce too much overhead.

By the modification of the migration procedure, the bottleneck of the serial migration of several components - the time needed to migrate one component before another migration could be carried out - was widened a lot. In Chapter 6 there is shown that the inaccessibility time of a component during migration[9] can be reduced to a fraction of time needed with the general migration procedure.

[9]Which is in fact the same time that is the bottleneck of concurrent migration.

Chapter 4

Load balancing

Chapter 3 discussed the process of migrating a component at runtime of the application. It was shown that component migration can be implemented into a component system (on the prototype implementation added to SOFA 2) and that migration can be transparent to all components of an application except for the migrated one, which needs to support some additional control functionality.

What needs to be discussed yet are the possibilities of utilizing the migration of components to improve system performance.

Since the thesis focuses mainly on the questions coming with the migration of components, this chapter only generally sums up the load balancing. The prototype implementation therefore focuses on providing a framework for helping with implementing sophisticated load balancing algorithms to separate them from the need of directly using the component migration.

4.1 Reasons for load balancing at runtime

4.1.1 Resource usage variances

When describing the components' layout among the Deployment Docks before launching the distributed application, full information about the computers within the distributed component system might not be available. If the application runs long enough, there might for example occur some changes in the network topology - e.g. new computers can be added which are not utilized by the system because their Deployment Docks were not known at deployment time. Another example might be that the computers the Deployment Docks are running on could be used by other applications for nontrivial computation tasks. This could lead to high cpu usage on machines that contain components that also generate heavy load on the processor. It would be therefore favorable to migrate them to computers with lesser load.

However, the requirements vary from system to system. It would be nearly impossible to design an algorithm that would befit all possible situations.

4.1.2 Migrating clients to servers when huge load of communication is anticipated

Another option is to utilize the migration of components to eliminate the latency caused by network communication between components which interact a lot with each other.

Which components are communicating a lot with each other and should therefore be migrated to the same location can be detected in a number of ways. The most simple method would be utilizing the connectors to monitor the number of method invocations per some given time quantum. In some systems (for example ProActive [19]) components are even given the option to migrate to another one. This possibility could be utilized by the developers of the application when they know that two components are going to communicate a lot with each other.

4.2 Monitoring system load in a distributed system

Monitoring the system resources usage is fundamental for balancing the load. It can be done on many levels - from the global view of the performance of the computers participating in the distributed system all the way down to the components themselves and the connectors they are interconnected with. For a good support of the implementation of load balancing algorithms, the component system should provide the means of analyzing the load on the different resource types (or the *load index*, as called for example in [8] or [9]).

4.3 Load balancing principles

Load balancing is a discipline which has been studied ever since the beginnings of distributed systems. Therefore, most of the known algorithms of load balancing are focused on the load balancing based on the migration of processes, not components. The basic principles are however applicable in both cases.

Load balancing algorithms can be either static or dynamic. Static algorithms utilize the knowledge gained by the analysis of the system before it was launched. The rules for load balancing are hard-coded in the static algorithms. Since they cannot adapt to the changes in the running system, the static load balancing algorithms are not very suitable for general purposes.

On the contrary, dynamic load balancing algorithms focus solely on utilizing the knowledge of the load of a running systems. According to [8], dynamic load balancing algorithms use the following four policies for making their decisions:

1. *Transfer policy* decides whether a node of a distributed system should be used for load balancing at the moment by either accepting new tasks (in which case

the node is called a *receiver*) or for handing over some of its tasks (in which case the node is called a *sender*).

2. *Location policy* has the responsibility of finding either suitable *receivers* in case of *sender* initiated transfers or *senders* in case of *receiver* initiated transfer.

3. *Selection policy* takes care of choosing which tasks should be moved from the *senders*.

4. *Information policy* handles the options of when the information about the state of the system should be collected, what types of values should be measured and where they should be measured. According to [8] there are three types of *information policies*: demand driven[1], periodic[2] and state-change driven[3] *policies*. The difference between demand driven and state-change driven *policies* is that while in demand driven *policies* a node collects data about its surroundings, in state-change driven *policies* it sends information about itself to the others.

4.3.1 Example

Even though the issue of distributing tasks among the nodes for an optimal load distribution is an NP hard problem, it seems that even simple heuristics can lead to acceptable results. For example, in [21] there is described a load balancing algorithm which uses constant thresholds (both for overloading and underloading of the nodes) on cpu utilization to decide whether a node becomes a *sender*, a *receiver* or remains neutral to the load balancing at the moment. The *selection policy* uses random targets and the *location policy* chooses the first suitable receiver found. The *information policy* is demand driven. Since the article demonstrates that such a load balancing algorithm is efficient enough to spread the load evenly between the computers, the prototype load balancing implementation of this thesis builds on the same principles. The details are described in Chapter 5.

[1] These are usually decentralized policies in which a node collects data about its surroundings only when it becomes either a *sender* or a *receiver*.

[2] These *policies* can be both centralized and decentralized. The state of the system is collected in time periods.

[3] These can be also both centralized or decentralized. Under state-change driven *policies*, a node reports the data about itself when its state changes by a given degree.

Chapter 5

Prototype implementation

During the work on the thesis there was developed a prototype implementation of both component migration and simple load balancing to prove that the concepts described in the thesis are applicable on a real component system. SOFA 2 was extended with transparent migration of components and tools for handling migration (including an extension of the MConsole which is, however, beyond the scope of this thesis and will not be described here).

This chapter sums up the changes which had to be to SOFA 2 and should therefore belong also in its documentation (available at [4]).

5.1 Changes which had to be made to SOFA 2

Most changes committed to SOFA 2 on behalf of this work were done to the implementation of the Deployment Dock and related classes and to the Connector Generator (congen). The following paragraphs describe the most important ones.

5.1.1 Exclusive usage of migrating connectors

The connectors supporting migration of components are crucial for the prototype implementation and their sole usage had to be enforced. The changes made to the connectors are not of fundamental base, so the migrable connectors are compatible in the previously used non-migrable ones. It would be unreasonable to combine the two types together though and non-migrable connectors have been declared deprecated.

Method invocation connectors

Method invocation connectors had to be implemented in such a way that they would provide both the local and remote communication types[1]. In order to obtain such

[1] Until now SOFA 2 distinguished between local and remote method invocation connectors. Their usage was decided with respect to the topology of components described in the application's Deployment Plan.

connectors, templates of hybrid connectors (as they started to be called) were added to the connector generator. They are based on the Remote variant of *Stubs* and *Skeletons* of the method invocation connectors. They utilize the fact that the *Remote Reference Bundles* can carry both local and RMI stringified references to connect locally if available[2].

As stated earlier in the thesis, connectors provide the migration functionality by correctly implementing the methods of the `ElementMigrable` interface. For method invocation connectors this means:

- *Stubs* use the `onBeforeMigration` method to store and send forward the `RemoteReferenceBundles` describing the location of the *skeletons* they are pointing at. The `onNewInstanceCreation` method then binds the remote references. The `onAfterMigration` method contains no code for the *stubs*.

- *Skeletons* on the other hand don't need to use the `onBeforeMigration` method. Instead, when the new instance is created, the *skeletons* store their `RemoteReferenceBundles` in the `onNewInstanceCreation` method which is then sent back to the old instance of the *skeleton*. There it is processed by the `onAfterMigration` method and set to be thrown as the data of the `MigratedException` which is used to notify the clients of the *skeleton* to reconnect to the new instance.

Messaging connectors

Since the messaging connectors are implemented by only one class describing both the connectors of the provided and required interfaces (`Mesg_send_recv`), the migration functionality had to be implemented wisely to avoid mixing the code needed by required connectors with the code needed by provided connectors.

SOFA 2 uses JMS [11] for the implementation of the messaging communication between components. Communication using a *topic* is used for all the cases because it allows general communication of `m` senders with `n` receivers. Thanks to this fact, the binding of the connectors to the *topic* is the same for both senders and receivers (i.e. provided and required interfaces).

The old implementation of messaging connectors used the JMS's `ObjectMessage` objects to pass the messages. All the required interface connectors connected to the topic then processed the message. Since the migration functionality needs to target specific instances during the migration, the implementation had to be extended. Objects of the type `SOFAMessage` are used for publishing into the topic now. They contain the information about the type of the message (whether it is a normal one or a message received during migration and now being resent to the new instance) and the target ID[3].

[2] The principle is described in Chapter 3.

[3] All messaging connectors now have a unique ID used for their addressing during migration.

In the code, the reconnection looks precisely like the reconnection of the method invocation *stubs*. In the `onBeforeMigration` method, the `RemoteReferenceBundle` describing how to connect to the topic is stored in the parameters. It is later used in the `onNewInstanceCreation` method at the new instance to bind to the topic. The required interfaces have to do some additional functionality though.

Since the communication of the messaging connectors is not delayed as with method invocation, the required interfaces of the old instance have to store all the messages received during the migration[4]. The switch of the required interface's mode from receiving to storing in a queue is done inside the `onBeforeMigration` method.

As the new instance is created, it also has to set its required interfaces in the `onNewInstanceCreation` method to postpone incoming messages until it receives all the messages that were sent during the migration. In the `onNewInstanceCreation` method the connector's ID is also passed to be sent back to the old instance which then uses it as the target of the resent messages. The `onAfterMigration` method at the old instance then takes care of resending those messages.

The Thread Observer

The connectors have to not forget to use the `ThreadObserver` of the component. Particularly, the `enterThread` and `leaveThread` methods should be executed whenever the connectors pass a synchronous call to the component. The `ThreadObserver's threadInside` method is then used for checking the cycles in the dynamic dependencies when the component is being stopped and the calls are delayed inside the connectors.

5.1.2 Deployment Dock changes

Deployment Docks contain and manage all the components within a SOFAnode and also contain additional information needed for controlling the components. It was more than natural that the Deployment Dock also had to take care of migrating the components.

Since the migration process takes place on two different Deployment Docks, it could not be handled within one method. The process was split up into two separate parts which are executed at the source and destination Deployment Dock. The called `emigrateComponent` takes care of the source side migration. During the `emigrateComponent` method the component is first asked to preinstantiate at the destination dock using the `preInstantiateComponent` method. When the method successfully returns back from the destination dock where the component is prepared, the component is stopped and its state is transferred to the destination where it is resumed using the `loadAndResumePreInstComponent` method.

[4]Otherwise these would be lost for the new instance and the component would never receive them.

At the appropriate parts of the migration process, the `onBeforeMigration`, `onNewInstanceCreation` and `onAfterMigration` methods are executed on all the component's interfaces and the `parameters` of the methods are transferred to be passed to be following calls.

The instantiation of components and the data structures describing them at runtime had to be altered a bit. They were changed to make it easier to distinguish between migrable components and to pass the necessary data needed by migration. The `MigrationInfo` and `MigrationResult` classes were also introduced to carry the data needed to be exchanged between the source and destination Deployment Docks.

5.1.3 Executing migration

For executing component migration from the command line, the class `MigrateComponent` was implemented. This class only has the `main` method which expects to be passed two arguments. The first argument is the id of the component which should be migrated. The second argument is the name of the destination Deployment Dock. The `main` method takes these arguments and calls the `emigrateComponent` method of the Deployment Dock which the migrated component resides in.

5.2 Load balancing

In the prototype implementation, the main class taking care of load balancing is the `MigrationManager`. It actually does not do almost any work itself. For making decisions about the migration of components to balance the load of the system, the `MigrationManager` has to be presented with an instance of a class implementing the `MigrationWorker` interface. The `MigrationManager` simply periodically asks the worker it has been presented with for new orders on component migration. If the returned list of orders is non-empty, the `MigrationManager` goes through the list and sequentially migrates the components to the required destinations.

For example purposes, two implementations of the `MigrationWorker` have been developed. The implementations have been developed for presentation purposes because they provide rapid generation of migration orders triggered by even a small disturbance of the equilibrium they try to maintain.

- **SimpleCPULoadBalancer**

  The `SimpleCPULoadBalancer` distributes the components among Deployment Dock depending on the load of the cpus the Deployment Docks are running on. At instantiation time, the `SimpleCPULoadBalancer` gets the threshold of the cpu load above which a Deployment Dock is regarded as overloaded. The second parameter passed when creating an instance is the ratio of the threshold value which marks a Deployment Dock as underloaded.

A random number of migrable components from overloaded Deployment Docks is ordered to be migrated evenly spread over the underloaded Deployment Docks.

- **SimpleDistributionBalancer**

  The `SimpleDistributionBalancer` does not monitor the system load at all. It just tries to spread components evenly among Deployment Docks with as little the deviation as possible. The maximum allowed deviation is passed as a parameter when instantiating the `SimpleDistributionBalancer`.

Chapter 6

Evaluation

The thesis describes a fully functioning process of migrating components in distributed hierarchical component systems. On the prototype implementation for SOFA 2 it was shown that the migration process described in the thesis is easy to implement with little changes made to the actual architecture of a component system. The concept of separating the migration intelligence of different communication styles exclusively in the software connectors has proved to be usable on the implementation of migration for the *method invocation* and *messaging* communication styles.

The following sections of this chapter show the measured differences between component inaccessibility time during migration when using the general migration procedure and the proposed preinstantiation based modification.

6.1 Migration time evaluation

6.1.1 Testing environment

In section 3.4 there was proposed a modification of the general migration procedure which should reduce the time at which a component is stopped[1] during migration. To experimentally prove the advisability of such a modification, two computers were set up. They had the following configuration:

- Core 2 Duo T9600 on 2.8GHz with 4GB RAM, running Ubuntu 10.04 64bit, further marked as **Computer A**
- Two AMD Opteron 244 processors on 1.8GHz with 2GB RAM, running Ubuntu 8.04 64bit, further marked as **Computer B**

The two computers were connected by 100Mbps ethernet. Computer A ran the `Zeroconf` server, the `DeploymentDockRegistry`, the `GlobalConnectorManager` and the SOFA 2

[1] And thus inaccessible and also delaying the application if there emerges a call on it.

Repository. Both the computers then ran a DeploymentDock. The SOFA 2 application developed for the experiment consisted of a modified SOFA 2 LogDemo. There were two components connected by method invocation communication style. On one side there was a tester component with one required interface sending requests to the provided interface on the logger component. The logger component was implemented by the class shown in Figure 6.1.

Subsequently, the logger component carrying no data, 1kB of data, 1MB and 10MB of data was migrated 20 times (10 times in each direction) each time first with the general migration procedure and then in another set with the improved one. Times of inaccessibility of each of the component's sizes on both procedures were noted down. The environment was restarted between the experiments on the different component sizes.

The components were simple implementations of a migrating component with one provided interface. The different data sizes were obtained by a field of the component class carrying an array of the type byte. The size of the array then was the size of the component's data in bytes. Figure 6.1 shows the code of the component. Note that for the component with no data, the field data was removed.

```
@Migrable
public class ExperimentLogger implements iface.Log {

  final int COMPONENT_SIZE = 1024; // or 1024*1024 or 10*1024*1024

  @PersistentAttribute
  byte [] data = new byte[COMPONENT_SIZE];

  public void log(String message) {
    System.out.println("LOG: " + message);
  }
}
```

Figure 6.1: The component used in the experiment for measuring migration times.

6.1.2 Results

When the times of migration of different sized component and using either of the migration procedures were measured, a mean value was calculated and the results were plotted in a graphic chart. The results themselves are shown in a table in Figure 6.3, the chart is shown in Figure 6.2.

Several conclusions can be drawn from the results. The main one which this section aimed for is, that separating the migration process into component preinstantiation (during which the old instance is still fully working) and the actual state

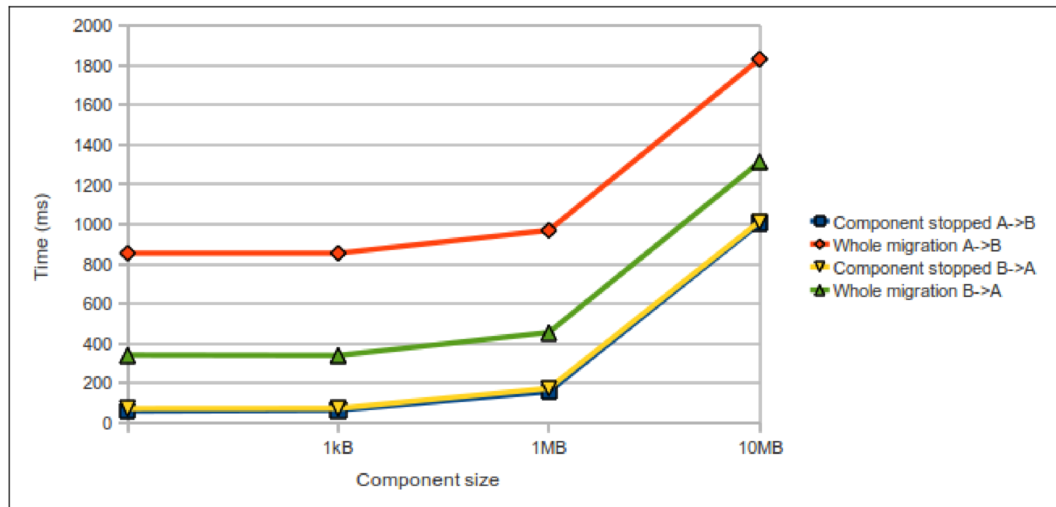


Figure 6.2: Measured migration times for different component sizes

	A → B		B → A	
Component size	Component stopped	Whole migration	Component stopped	Whole migration
0	60 ms	855 ms	73 ms	342 ms
1kB	64 ms	855 ms	74 ms	340 ms
1MB	158 ms	970 ms	175 ms	456 ms
10MB	1004 ms	1832 ms	1015 ms	1314 ms

Figure 6.3: Table of values shown displayed in Figure 6.2 chart

transfer, is profitable for the whole process. From the results we can see, that the size of the inner state of a component influences the migration (and inaccessibility) time noticeably only when its size is more than several kilobytes. Even at the inner state size of 1MB, the inaccessibility time of the improved migration procedure is several times shorter.

An interesting thing is, that the inaccessibility time of the unimproved procedure is a lot different when migrating from Computer A to Computer B from when migrating a component from Computer B to Computer A. The difference is caused by the fact that Computer A contained all the SOFA 2 infrastructure - mainly the `Repository`. Since the communication rate with the `Repository` is quite high during component instantiation, migration is mainly slowed down by this part. Therefore, when the component was migrated to Computer A, which could communicate locally with the `Repository`, since it was running it, the whole migration time was a lot shorter.

Even though the inaccessibility time when using the unimproved migration procedure differs so much, the measured inaccessibility time when using the improved migration procedure is about the same. The about 10 millisecond difference that

occurred by all the component sizes can be caused by differences in both hardware and software of the two computers.

Another notable fact is that the difference between the inaccessibility time for the improved migration procedure and unimproved one is (in the same direction) the same no matter what is the size of the inner state of the component. This means that the size of the component's inner state only influences the inaccessibility time of the migration.

Chapter 7

Related work

The process of migrating components described in this thesis is of course not the only possible way. There have been published several papers for component migration. In this chapter we focus on examples of migration in CORBA and ProActive and compare it to the solution proposed and implemented in this thesis.

7.1 Migration in CORBA

In [16] there is proposed a solution for the migration functionality based on the *Lifecycle Service* (described in [17]). According to it, CORBA objects, which want to be manageable, have to implement the `LifeCycleObject` interface that also defines the `move()` method. Since CORBA itself is not a component system and contains no containers like Deployment Docks which would take care of object instantiation, the creation of objects needed by the migration functionality is done using *factories*[1]. The system utilizes a central registry for factories which can be queried using a *factory finder*. The migration functionality does not modify the ORB but simply builds on top of CORBA.

The first major difference to the solution proposed in this thesis is, that an object can be requested to migrate by "anyone" - starting with an outside entity just as with the migration implementation in SOFA 2, but also by the client that uses the proxy to the object and even by the migrable object itself. This choice seems logical. Since there is present no Deployment Dock-like container that would present additional location abstraction (only the *factories*, but these primarily take care of object instantiation), there is no reason not to give the migrable objects full control. It also gives the developers using the migration functionality more possibilities.

However, the choice to present the components in SOFA 2 with the possibility to initiate migration of themselves (or other components as well) would be against its basic philosophy. SOFA 2 aims at such a high level of distribution transparency that component code contains no sign of distribution. The components have no way of

[1] Implemented as CORBA objects which can create other objects.

discovering whether the opposites they are connected to are within the same process or at the other side of the Earth. This information is not even available at the time of development of the components.[2]

The article then proposes two mechanisms of the migration process. A so called *passive migration* and a *state sharing migration*. The difference between these two is in the handling of the incoming calls which occur during object migration. The *passive migration* is mostly the same as the mechanism described in this thesis. It pauses incoming calls which emerged after the request to migrate the component and waits with the migration until the ongoing requests are processed. However, the paper does not handle cycles in the dynamic dependencies of the delayed invocations. In this case the solution of this thesis seems better thought out.

The *state sharing migration* aims at reducing the time during which the object is unavailable. When an object is migrated, the mechanism creates a new instance at the destination site, but the state of the object is not transferred. Instead, a remote reference to the state is passed to the new instance. When there emerge new invocations on the object, they are directed at the new instance which can handle them with no delay. The state of the object is transferred to the new instance no sooner than when all ongoing invocations laid on the old instance are processed. The incoming calls of course have to be delayed during the state transfer.

There is no doubt that the *state sharing migration* mechanism spares time when the unavailability period of the migration is concerned. The question is, whether it repays the fact that the object has to access its state remotely for a nontrivial time period. This would depend on a particular situation.

Similarly to the solution proposed by this thesis, the relocation transparency is handled by a client-side proxy object. The reconnection of the remote references utilizes a naming service to rebind the references when their old values become invalid. Unlike the solution of the thesis, the article does not take care of the possibility of reconnecting locally to objects which might have been migrated to the same address space as where the clients reside.

7.2 Migration in ProActive

One of the well known systems for distributed computing is ProActive [19]. It supports object migration in its default implementation, and thus it is more than interesting to compare the way ProActive migrates objects with the way this thesis does.

Just like SOFA 2, ProActive is in fact a Java framework with no modifications to the Java runtime. Java objects in ProActive are separated into *active objects* and *passive objects*. The *active objects* are the ones which can be transparently distributed. They consist of a so called *body* and the standard object. The *body* is similar to SOFA 2's *skeletons*. *Passive objects* are normal java objects. When

[2] As described in [2], the distribution of components is set when creating the Deployment Plans.

methods with parameters are invoked on an *active object*, *active object* parameters are passed by reference and *passive object* parameters are passed by deep copy.

Furthermore, ProActive presents a model of an application which is structured into so called subsystems. Each *active object* belongs to a particular subsystem and each subsystem has just one *active object*. The subsystems resemble a lot components as they are understood by SOFA 2. There can be no shared *passive objects* between several subsystems and if an *active object* has a reference to a *passive object*, it means they are in the same subsystem. Only the *active object* is visible to the outside of the subsystem.

Active objects are hosted inside so called *nodes*. The *nodes* are containers for *active objects* and provide location transparency in the same way as SOFA 2 *Deployment Docks* do.

Even though there are the similarities described above, some ProActive concepts differ from those of SOFA 2 in fundamental parts. First of all, any object can be made active, the decision can be made anytime. Another one, and probably the most fundamental, is that each subsystem (and thus each *active object*) has its own thread which handles all invocation made on the *active object*. All the calls made on the *active object* are processed by its *body* which stores them in a queue of pending calls. No additional parallelism is allowed in an *active object*. The last difference important for the sake of this comparison are the *future objects*. When an invocation is made on an *active object*, the invocation at the client side returns instantaneously. If the invoked method returns a value, the instantaneous return without waiting for the method to run (not to mention waiting for its return) means that the return value cannot contain the actual method's result. Instead, a *future object* is created which can be seen as a proxy. When the *future object* is really accessed for the data, it waits for the method to finish if it has not done so already. ProActive is said to use asynchronous calls with automatic synchronization.

Just as the CORBA example mentioned above and unlike the proposed solution for SOFA 2, ProActive provides the *active objects* with the option to migrate themselves. Migration can be also triggered from the outside. The possibilities of aiming the migration to a destination are either migrating to a specified *node* or to a specified *active object*. An object that wants to migrate must call one of the `ProActive.migrateTo(...)` methods and the call has to be the last call of the method which uses migration. External agents can migrate an *active object* by calling one of the `ProActive.migrateTo(...)` methods with the *active object* as a parameter.

The concepts of ProActive give it many advantages when migration is concerned. When a ProActive *active object* is being migrated, the system first suspends its execution. Since the *active object's* calls are stored within the queue of pending calls, it can be simply given no new calls from the queue. Moreover, because there is always only one thread of execution, it means there is just one call being executed at the *active object* at a time. The suspending mechanism can therefore just wait for this one call to finish.

Another notable fact is that ProActive does not have to take care of dynamic dependencies among the suspended calls. This is caused by calls on *active objects* being asynchronous and the usage of the *future objects*. If there would be a cycle in the dynamic dependencies on a call from a currently migrated *active object*, the incoming call would be also simply put in the queue. Thanks to the synchronization being lead through only when the results of the call are needed, the *active object* has enough time to migrate before processing the request.

When the state of the *active object* is transferred to the new location, the whole subsystem is transferred using deep copy. It comprises also the copy of all the pending calls from the queue, all the *passive objects* and *future objects*. As the *active object* is reconstructed at the new node, references to and from other *active objects* are updated and changes of local and remote references are taken care of. For calls invoked on the *active object* during its migration, *future objects* are utilized.

The way migration is implemented in ProActive is doubtlessly very elegant and in some ways more simple than the solution of migration implemented with this thesis for the SOFA 2 system. Some things about ProActive might be however too restrictive - for example the condition that there is just one thread taking care of an *active object*. The way migration is implemented in ProActive is also very specialized and would be rather hard to implement in a component system without turning it into a second ProActive. On the contrary, SOFA 2 and the prototype implementation of migration developed for this thesis put no limitations on threads inside a component. The concept of migration described in the thesis aimed to be as general as possible.

Chapter 8

Conclusion and future work

8.1 Summary

The goal of this thesis was to analyze the problem of migrating components in a component system and to propose and implement a working solution. Throughout the thesis there was shown that by following some basic principles, component migration can be easily implemented into a complex hierarchical component system that SOFA 2 doubtlessly is.

The thesis focused mainly on the migration of components in a component system and problems that can be encountered during its implementation. The issues have been analyzed and their possible solutions have been described. The prototype implementation developed with the thesis proves that the proposed solutions are applicable enough to provide a means of transparent migration of components with seamless integration to an already developed component system. When developing migration enabled components in SOFA 2 only little additional code is required. It was also shown that the principles described in the thesis are general enough to be used to provide migration in other component systems as well.

Since load balancing is absolutely dependent on the migration of components, the thesis focused purposely mainly on the migration part. A simple framework for component migration was developed for the possibilities of easy adding load balancing implementations later.

8.2 Future work

The scope of load balancing was left behind a little in this thesis. Possibilities of utilizing the knowledge of the components and measuring their resource consumption could lead to interesting load balancing algorithms.

One of the bottlenecks of the proposed solution for component migration can be that the proposed system does not support concurrent migration of a larger number of components. To make it available, the deadlock checking functionality

should be improved to provide intelligence needed to analyze possible cyclic dynamic dependencies between migrated components.

It is clear that when migrating components with a large inner state, the migration can take quite a long time. The possibilities of synchronizing the component's state and its differences and other possible means of transferring the component's state faster should be also analyzed.

Bibliography

[1] Bures, T., Hnetynka, P., Plasil, F.: SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model, Proceedings of SERA 2006, Seattle, USA, Aug 2006

[2] Bures, T., Hnetynka, P., Plasil, F.: Runtime Concepts of Hierarchical Software Components, In International Journal of Computer & Information Science, Vol. 8, No. S, ISSN 1525-9293, pp. 454-463, Sep 2007

[3] Bures, T., Malohlava, M., Hnetynka, P.: Using DSL for Automatic Generation of Software Connectors, In procceedings of ICCBSS'08, Madrid, Spain, IEEE CS, Feb 2008

[4] SOFA 2, http://sofa.ow2.org/

[5] OMG (2008): OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2, http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF/

[6] Balek, D., Plasil, F.: Software Connectors and Their Role in Component Deployment, Proceedings of DAIS'01, Krakow, Kluwer, Sep 2001

[7] Chen, X., and Simons, M. A component framework for dynamic reconfiguration of distributed systems. In Lecture Notes in Computer Science, Volume 2370 (Jan 2002), vol. 2370.

[8] N. G. Shivaratri, P. Krueger, and M. Singhal. Load distributing for locally distributed systems. Computer, 25(12), 1992.

[9] K. Chow and Y. Kwok. On Load Balancing for Distributed Multiagent Computing. IEEE Transactions on Parallel and Distributed Systems, 13(8):787–801, August 2002

[10] Java SE 6 Documentation, http://java.sun.com/javase/6/docs/

[11] JMS documentation, http://java.sun.com/products/jms/

[12] Apache Ant, http://ant.apache.org/

[13] The Ivy dependency manager for Ant, http://ant.apache.org/ivy/

[14] M. Henning, "Binding, Migration, and Scalability in CORBA," Communications of the ACM special issue on CORBA, vol. 41, Oct. 1998.

[15] D. Janaki Ram, A. Vijay Srinivas, Object migration in CORBA, J. Comput. Soc. India 32 (1) (March 2002) 18–27.

[16] Peter, Y. and Guyennet, H. 2000. Object mobility in large scale systems. Cluster Computing 3, 2 (Apr. 2000)

[17] Common Object Services Specification, Vol. I, Technical Report 94-1-1, Revision 1.0, Object Management Group (March 1994)

[18] Xu B., Lian W., Gao Q. Migration of active objects in ProActive (2003). Information and Software Technology, 45 (9), pp. 611-618

[19] ProActive, http://proactive.inria.fr/

[20] Arregui D., Pacull F., Willamowski J., Rule-Based Transactional Object Migration over a Reflective Middleware, Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, p.179-196, November 12-16, 2001

[21] Javier Bustos-Jimenez , Denis Caromel , Alexandre di Costanz , Jose M. Piquer, Balancing Active Objects on a Peer to Peer Infrastructure, Proceedings of the XXV International Conference on The Chilean Computer Science Society, p.109, November 07-11, 2005

[22] Fractal, http://fractal.ow2.org/

[23] Enterprise Java Beans, http://www.oracle.com/technetwork/java/index-jsp-140203.html

[24] Microsoft DCOM, http://www.microsoft.com/com/default.mspx

[25] Amoeba, http://www.cs.vu.nl/pub/amoeba/

[26] Sprite, http://www.eecs.berkeley.edu/Research/Projects/CS/sprite/sprite.html

[27] Mosix, http://www.mosix.org

Appendix A

Contents of the enclosed disc

A.1 Structure of the disc

- `/tex`
  The `/tex` directory contains the sources of the thesis in LaTeX. It also contains all the images displayed in the thesis. They are stored in the directory `/tex/images`.
- `/bin`
  The `/bin` directory contains the binaries of the examples for easy launching.
  - `/bin/sofa`
    The `/bin/sofa` directory contains a prepared SOFA 2 environment including a pre-filled repository.
  - `/bin/mconsole`
    The `/bin/mconsole` directory contains prepared MConsole environments for several architectures: Linux x86, Linux x86_64, Windows x86 and Windows x86_64.
- `/src`
  In the `/src` directory there are the full sources of SOFA 2 extended with migration which are needed to compile the examples.
- `/master_thesis.pdf`
  The pdf in the root of the disc contains this text in an electronical form.

A.2 Examples howto

A.2.1 System requirements

The examples provided with this thesis run (were tested) on 32bit or 64bit GNU/Linux and 32bit Windows with Java 1.6.

To compile the sources included with the thesis, JDK 1.6[10] is required along with Ant[12] and Ivy[13]. The build can be initiated by running ant in the `build` directory. The repository is refilled by running ant in the `sofa-j/trunk/dist` directory.

A.2.2 Running the examples

The scripts needed for running the included demos are located at `/bin/sofa/bin`. Since SOFA 2's logging system requires write access to the `/bin/sofa/log` directory, the contents of the `/bin/sofa` directory should be copied to the harddisk first. Windows users then have to set the environment variable `SOFA_HOME` to the location of the contents, otherwise SOFA 2 will not launch.

Running the examples needs several of the scripts to be launched in correct order. All the scripts come both in manually configured and automatically configured versions. It is strongly recommended to use the autoconfigured scripts, especially when running SOFA 2 distributed among a number of computers. Windows users use the *.bat files (or the *.sh scripts from Cygwin) while GNU/Linux users should use the *.sh files.

The following scripts need to be launched correctly in order to run the examples.

1. `sofa[-auto]-node.(bat/sh)`

   This script is an essential one. Almost every action which SOFA 2 provides requires it to be launched. It starts the GCM, DDR and Repository. The *auto* version of the script also launches the Zeroconf server needed by all the other *auto* scripts.

2. `sofa[-auto]-dock.(bat/sh) dock-name`

   The Deployment Docks have to be run to host the components of SOFA 2 applications. They are typically called `nodeA`, `nodeB`,...

3. `sofa[-auto]-launch.(bat/sh) deployment-plan -v version`

   The launch script is used for executing the SOFA 2 application. It needs the full name and version of the Deployment Plan describing the application to be passed as its parameters.

4. `sofa[-auto]-migrate.(bat/sh) component-id new-dock`

   This script is used to manually migrate components between the Deployment Docks. Be sure to pass a correct id of a migrable component, otherwise the script will report an error.

   The `component-id` is the internal id of the component in form `dockname@number` - e.g. `nodeA@1`. The `new-dock` means the name of the Deployment Dock which is going to be the new destination of the component.

5. `sofa[-auto]-migration-manager.(bat/sh) {options}`

   The `sofa[-auto]-migration-manager` script is used to run the sample load balancer. The example migration manager provided at the disc runs the simple distribution worker and thus needs only one parameter - the allowed difference of the number of components between the deployment docks.

6. `migration_demo_(rmi/messaging).(bat/sh)`

   These scripts were introduced for easier launching of two of the demo migrable applications. They can be used instead of the `sofa[-auto]-launch.(bat/sh)` script.

For a more detailed information about the usage of SOFA 2, refer to [4].

A.2.3 List of the example Deployment Plans

The prefilled SOFA 2 repository provided at the disc contains the standard SOFA 2 logdemo, newly added migrable logdemos and messaging demos, which are also migrable.

- `org.objectweb.dsrg.sofa.examples.logdemo.deplplan.Local`
- `org.objectweb.dsrg.sofa.examples.logdemo.deplplan.Distributed`
- `org. ... .deplplan.Composed[Local/Distributed]`
- `org. ... .deplplan.Migrable[Local/Distributed]`
- `org. ... .deplplan.MigrableForwardingLogdemo`
- `org. ... examples.messaging.deplplan.MessagingDemoLocal`
- `org. ... examples.messaging.deplplan.MessagingDemoDistributed`

Launching of these demos requires one Deployment Dock called `nodeA` to be running for the `Local` variants and three Deployment Docks called `nodeA`, `nodeB`, `nodeC` for the `Distributed` variants of the SOFA 2 applications.

To list all the entities that are stored in the repository, cushion can be used by running `cushion print all`.

The easiest way of launching the deployment plans is probably using the MConsole. In the list of the deployment plans in the repository, simply right click the deployment plan intended to run and select `Run As->SOFA 2 application` in the context menu. The SOFANode and Deployment Docks have of course to be running.

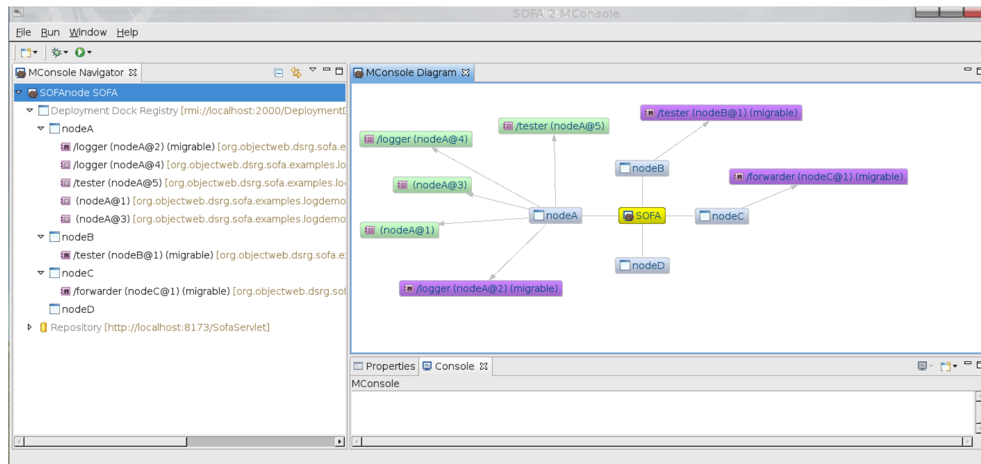


Figure A.1: Example MConsole

A.2.4 Migration in MConsole

For a more illuminating view of the migrable components, the standard MConsole of the SOFA 2 was extended with visually distinguishing migrable components from non-migrable ones. The MConsole also contains the possibility of migrating the components by dragging them in the navigator toolbar.

Since MConsole is an Eclipse plugin, launching it requires the same as launching Eclipse. In the directory `/bin/mconsole` there are archives with an Eclipse distribution for either 32bit GNU/Linux, 64bit GNU/Linux or 32bit Windows. To install it, simply copy it to your disc and extract the archives. They contain a prepared Eclipse environment with the MConsole plugin.

Before the MConsole can be used to migrate the running components, it must be first set up to connect to a running SOFAnode. If you have a running SOFAnode (preferably an autoconfigured one), select `File->New Wizards->SOFAnode`. A dialog for configuring the SOFAnode connection should pop up. Enter the name you wish to use for your SOFAnode, select the Autoconfigured connection and click `Finish`.

Figure A.1 shows a sample MConsole with a configured SOFAnode with four Deployment Docks and several running components. As can be seen in the diagram view, normal components are painted in a pale green color while migrable components are distinguished using a tone of purple.

To migrate components using the MConsole, simply drag a component marked as migrable in the MConsole Navigator and drop it into a Deployment Dock in the navigator. The MConsole should then ask if the migration should be executed. When it is done, the navigator tree will collapse the open branches and needs to be open again to view the changes. To refresh the diagram view, click the SOFAnode in the navigator. Note that the diagram view cannot be used for component migration.

For a more detailed description on the usage of the MConsole, refer to [4].

A.3 Known issues

There are several problems not caused by the prototype implementation which need to be mentioned. The problems might occur depending on which system the implementation is executed on. Most of them can be successfully bypassed.

- **On MS Windows, the log4j does not find the target directories where the logs should be saved.**

  This problem might happen when launching the Deployment Docs and spams the execution console. It does not however affect the runtime.

- **Distributed execution does not work on GNU/Linux.**

  This might be caused by the hostname of the computer in `/etc/hosts` being set to `127.0.1.1`. To solve the problem, change it to the computer's IP address.

- **Build of sofa-j/dist ends with errors without publishing entities into the SOFA 2 repository.**

  It can happen that the repository does not start in time to accept requests generated by the build process. Try until it goes through successfully.

- **MS Windows cannot compile the sources of SOFA 2.**

  This might happen when there are several versions of Java installed on the system. Uninstall all of them and make a clean install of JDK1.6.

- **Distributed execution does not work when using the messaging demos.**

  This is a problem of SOFA 2 itself, not the thesis. A bugfix should be coming soon.