Master’s Thesis

Replication and Migration of OSGi Bundles in the Virtual OSGi Framework

by

Damianos Maragkos

January 7th, 2008 - July 6th, 2008

Advisor:
Jan S. Rellermeyer
Prof. Gustavo Alonso

ETH Zurich
Department of Computer Science
8092 Zurich
Switzerland
Abstract

The OSGi framework is a Java-based technology which solves the problem of software modularization. Services are defined through a service interface and modularization is achieved with the use of bundles interacting through services. The Virtual OSGi Framework project explores the large-scale distribution of OSGi services over heterogeneous computer networks. OSGi bundles and services can be deployed to a virtual framework, which autonomously allocates them to physical nodes. Such an environment requires fault-tolerance, high availability and balanced load in order to be functional and resilient to failures which can occur in distributed systems. This can be achieved with the use of service replication and migration. The setup is highly dynamic and it reacts to changes in the utilization and availability of resources. The dynamism of the environment imposes fluid replication mechanisms, which place a replica and not a copy of a service, wherever it is needed.

In this thesis the replication and migration mechanisms of the Virtual OSGi Framework are presented and applied in real-world scenarios. The replication mechanism allows threads in a cluster of JVMs to interact with each other across the boundaries of a single virtual machine. It thereby extends the ideas of the OSGi standard and the Java VM to have a cluster-wide meaning. The clustering behavior is injected into the bytecode of the application classes at load time, without the need of changing the application. The migration mechanisms are applied at the application level providing portability at the cost of minor overhead. Thread state capturing code is injected at load time too, in order to attach a backup object to every thread which will contain the thread state in a serializable form.
Preface

This thesis is a part of the Virtual OSGi Framework research project of the Systems Group at the Department of Computer Science, ETH Zürich. It is submitted for the partial fulfillment of the Master Program (MSc ETH) in Computer Science. The duration of the thesis is 6 months, from January 7th to June 6th, 2008 in the Informations and Communications Systems Group under the supervision of Jan S. Rellermeyer and Prof. Gustavo Alonso.

Acknowledgements

First of all, I would like to thank my supervisor Jan S. Rellermeyer for offering me this challenge during my Master Thesis. I am very grateful for his quick responses and solutions provided during difficult parts of the project. I would also like to thank my mentor, Prof. Gustavo Alonso, for his support during the whole Master Program at the ETH Zürich. Eventually, this master thesis would not have been completed without the invaluable support of my family.
# Contents

Preface

1 Introduction
   1.1 Motivation ........................................... 1
   1.2 Related Work ......................................... 2
   1.3 Outline of the Thesis ................................. 3

2 OSGi
   2.1 OSGi Framework ....................................... 5
       2.1.1 Bundles .......................................... 6
   2.2 OSGi Services ......................................... 7
   2.3 The Virtual OSGi Framework ........................... 8

3 State capturing and Bytecode engineering .............. 11
   3.1 Introduction - What is state? ........................ 11
   3.2 Root objects and state definition ..................... 14
   3.3 Symbolic code analysis ............................... 15
       3.3.1 Overview ....................................... 15
       3.3.2 Symbolic analyzer ............................... 15
   3.4 Replication constructs ............................... 19
       3.4.1 Transactions .................................... 19
       3.4.2 Updates ......................................... 22
       3.4.3 Referenced objects ............................... 23
   3.5 Injection of Clustering Behavior–Technical Description .... 23
       3.5.1 Service object method ......................... 25
       3.5.2 Other methods ................................ 29
       3.5.3 State crossing the boundary of the service interface . 34

4 Replication of services ................................. 37
   4.1 Related Work ......................................... 37
       4.1.1 Terracotta ...................................... 37
   4.2 Virtual OSGi Framework replication implementation .... 39
       4.2.1 System model .................................... 40
       4.2.2 Service Replicas Coordination .................. 43
       4.2.3 Active object pattern ........................... 44
       4.2.4 Distributed synchronization ...................... 48
       4.2.5 Updates propagation ............................. 52
       4.2.6 Conflict resolution .............................. 54
## 5 Migration of Services

5.1 Java thread internals .......................................................... 58
  5.1.1 The Program Counter ................................................. 58
  5.1.2 The Java Stack ............................................................ 58
  5.1.3 The Stack Frame .......................................................... 58

5.2 Related Work ................................................................. 60
  5.2.1 Brakes ............................................................... 62

5.3 Virtual OSGi Framework Migration Implementation .............. 62
  5.3.1 Overview and design decisions ..................................... 62
  5.3.2 Implementation of thread serialization ....................... 64

## 6 Evaluation

6.1 HTTP Session replication .................................................. 79
6.2 Experimental environment ................................................ 79

## 7 Replication through VM Support

7.1 JVM Tool Interface ........................................................ 83
  7.1.1 Class Redefinition ..................................................... 84
  7.1.2 JVMTI Events ............................................................. 84
  7.1.3 Heap analysis ............................................................ 85

7.2 Virtual OSGi Framework Native Replication ....................... 86
  7.2.1 Watched Fields and Object Tagging ............................. 86
  7.2.2 Class Redefinition ..................................................... 86
  7.2.3 Access/Modification Callbacks ................................... 87

7.3 Performance Evaluation .................................................. 88

## 8 Conclusions

8.1 Future Work ................................................................. 92

## A Evaluation of the Java implementation

## B Evaluation of the native implementation
Chapter 1

Introduction

1.1 Motivation

The OSGi framework is a solution to the problem of software modularization. However, OSGi frameworks are isolated units, sometimes interconnected to enable development of distributed applications. In this thesis, the OSGi framework obtains new properties. The goal of the Virtual OSGi Framework project is to treat the whole network as a large virtual OSGi framework where services can be installed and used without explicitly building the applications around a fixed topology. The distribution of OSGi services over heterogeneous computer networks puts in risk the availability and efficiency of the services. For this reason, the framework autonomously and dynamically controls where to place bundles, when and which services to replicate. This technique potentially increases the availability and fault-tolerance of OSGi services. In order to interconnect the bundles running on different machines two techniques are used, R-OSGi [21] and replication of bundles, including their internal state. The goal of this thesis is to elaborate different techniques to migrate and replicate OSGi bundles over different machines. The replication and migration are transparent, which means that the applications remain unmodified and run as if they were meant to run normally. Replication is a key to providing high availability and fault tolerance in the system. A service’s performance can be improved by balancing the load among many replicas. Since the system is intended to extend the idea of OSGi service, replication transparency is required. The service interfaces to the outer world remain unmodified too and the consumers of the services are unaware of the fact that the services run replicated. A consumer should be able to access any available replica of a service at any time, therefore the replicas should remain consistent. Migration enhances service performance too, because it can replace remote calls to the service with local ones by migrating the service to the calling location. It is also a challenge to keep the performance of the applications as close to the performance during operation without replication and migration mechanisms. In order to evaluate the system, the replication techniques had to be applied to a real world application to explore the overhead of a service operating replicated.
1.2 Related Work

The Rover toolkit [3] combines relocatable dynamic objects (RDOs) and queued remote procedure calls for mobile applications. A relocatable dynamic object is an object with a well-defined interface that can be dynamically loaded into a client computer from a server computer (or vice versa) to reduce client-server communication. This technique allows for disconnected operations. However, every object is associated with a 'home' server and has to export all the local updates to it. Until then the RDO is in a tentatively committed state. The RDO is in a committed state only when the server has committed the updates and resolved the possible conflicts, using a conflict resolution functionality provided by the application. Consequently, the RDOs have primary copies on the server and secondary copies on the client.

Grid computing [4] can be regarded as the creation of a "virtual supercomputer" composed of a network of loosely-coupled computers, acting in concert to perform very large tasks. This technology has been applied to computationally-intensive scientific, mathematical, and academic problems through volunteer computing. Therefore, appropriate programming and runtime solutions are needed in order to write, deploy and execute applications on such heterogeneous distributed hardware in an effective and efficient manner. However, the network in the Grid is basically static and not as flexible and fault-tolerant as with the use of structured overlay networks.

In the context of mobile databases, the Bayou system [11] lets mobile clients actively read and write shared data. It looks to users like a centralized, highly-available database service. Its architecture is based on a division of functionality between servers and clients. A server is any machine that holds a complete copy of one or more databases. Clients are able to access data residing on any server with which they can communicate, and conversely, any machine holding a copy of a database, including personal laptops, should be willing to service read and write requests from other nearby machines. Portable computers can be servers for some databases and clients for others. A read-any/write-any replication scheme is used to maximize the user’s ability to read and write data. As a result, the replicated databases are only weakly consistent. All copies of a database are converging towards the same state and will eventually converge to identical states if there are no new updates. Instead of using a consensus algorithm to commit the updates, each Bayou database has one distinguished server, the 'primary', which is responsible for committing writes. The other, 'secondary' servers tentatively accept writes and propagate them towards the primary. Clients can read tentative data with an expectation that it will be committed with the same effect, if possible. In other words, Bayou provides application-controlled inconsistency, with application-specific conflict decision. In [13], the problems of update anywhere-anytime-anyway transactional replication are stated. Eager replication keeps all replicas exactly synchronized at all nodes by updating all the replicas as part of one atomic transaction (the original transaction). There are no concurrency anomalies, but replication reduces update performance and increases transaction response times because extra updates and messages are added to the transaction. Lazy replication algorithms asynchronously propagate replica updates to other nodes after the updating transaction commits, typically as a separate transaction for each node. The replication models are classified to group and master models, too. Group mod-
els allow updates at any node with a copy of the data (update anywhere), while in master model, only the master can update the primary copy of the object. All other replicas are read-only. Eager and lazy-master systems are the safest ones, but include deadlocks and hurt performance. Lazy group replication does not include deadlocks but needs reconciliation mechanisms. A two-tier replication model is introduced which resembles lazy-master model but allows the existence of tentative data. However, tentative data is not acceptable by all applications.

1.3 Outline of the Thesis

Most of the effort during this thesis was spent on fine-granular instrumentation of applications’ bytecode. Such a task provides to someone a very deep overview of the JVM and how it operates. The rest of the thesis takes a fresh look at well known problems in distributed systems, like replication, multicast, consistency, locking, consensus etc. Chapter 2 provides some general information about OSGi and the big picture of the Virtual OSGi Framework. Chapter 3 deals with the definition of the requirements for making an application run replicated without the use of a special API and the mechanisms used to achieve this. Chapter 4 introduces the general architecture of the system after the bytecode instrumentation at load time and the coordination of the replicas. Chapter 5 provides the mechanisms used to capture a thread’s internal state and implement thread serialization. Chapter 6 gives an example of where this system can be used in a real-world scenario and compares it with related implementations. Chapter 7 describes and evaluates an alternative native implementation of the replication mechanism. Finally, Chapter 8 concludes the thesis and describes possibilities for future work.
Chapter 2

OSGi

The OSGi Alliance [20] (formerly known as the Open Services Gateway Initiative) is an open standards organization founded in March 1999. The Alliance and its members have specified a Java-based service platform that can be remotely managed. The core part of the specifications is a framework that defines an application life cycle management model, a service registry, an execution environment and modules. Based on this framework, a large number of OSGi Layers, APIs, and Services have been defined.

2.1 OSGi Framework

The Framework is the core part of the OSGi specification. It provides a general-purpose, secure and managed Java framework that supports the deployment of extensible and downloadable service applications known as bundles. OSGi bundles can be dynamically downloaded, installed and removed when they are no longer needed by OSGi-compliant devices. Each bundle can register a number of services at the framework. The services can be shared with other bundles. The OSGi framework controls the installation of bundles and manages the dependencies between bundles and services. OSGi provides a concrete improvement to Java weak model of modularity. The framework provides a concise and consistent programming model for Java bundle developers, simplifying the development and deployment of services by de-coupling the service’s specification (Java interface) from its implementation. Bundle developers can bind to services solely from their interface specification and the selection of the implementation can be done during runtime. In this service-oriented interaction, service providers publish service descriptions and service requesters discover services and bind to the service providers. Publication and discovery are based on a service description. In the context of OSGi, a service is described as a Java interface, the service interface, along with a variable number of properties, the service properties, that are name and value pairs. Service properties differentiate service implementation with the same interface. The centralized service registry of the OSGi framework helps service providers to be discovered through queries formulated in an LDAP syntax.
2.1.1 Bundles

The bundles are the entities used to deploy Java applications at the OSGi Framework. The bundle is deployed as a Java ARchive (JAR) file. JAR files are used to store applications and their resources in a standard ZIP-based file format. The JAR file contains code and resources which can provide components to other bundles, called services.

A bundle is a JAR file that:

- Contains code and resources to implement the provided services.
- Contains a file that contains information about the installation of the bundle, the manifest file.
- Explicitly states all the dependencies of the bundle. The OSGi Framework has to resolve the dependencies in order to correctly install and start the bundle.
- Contains a special class called BundleActivator. This class has to be instantiated by the framework in order to call its start and stop methods, to start and stop the bundle respectively. Once the bundle is started, its functionality is deployed at the framework and its services are available to other bundles installed in the OSGi Framework.

The OSGi framework provides mechanisms to support continuous deployment activities. Deployment activities are realized according to a well defined series of states depicted in figure 2.2. The activation or de-activation of a physical bundle results in the creation or destruction of a unique logical bundle, materialized by an instance from a class inside the bundle called a bundle activator. When the instance is created, the execution environment calls an activation method that signals the logical bundle that it is active. When the physical bundle is de-activated, the execution environment calls a de-activation method. When the logical bundle is active, it can publish or discover services and bind with other bundles by accessing the framework’s services registry. A bundle may be in one of the following states:

- INSTALLED The bundle has been installed successfully.
- RESOLVED All the Java classes that the bundle needs are available.
- STARTING The bundle is being started, i.e. the start method of BundleActivator has been called.
2.2 OSGi Services

As mentioned before, the bundles are built around a set of services, that are available from a centralized service registry. All OSGi services are defined by a service interface and implemented as a service object. All dependencies are handled by the OSGi Framework. For example, when a bundle is stopped, all the services registered at the framework by that bundle should be automatically unregistered. The OSGi Framework provides a mapping between services and their service objects. It supports queries for finding a service that fulfills certain requirements and an event mechanism, so that bundles can receive events about certain service objects.

In general, registered services are referenced through ServiceReference objects. This is done because a bundle might want to know about a service without requiring the service object itself. A ServiceReference is passed to bundles without the implication of dependencies. If a bundle wants to use this service, it can obtain it by using BundleContext.getService(ServiceReference). A ServiceReference contains the properties and meta information of the service. A service interface is the specification of the service's public methods. A service object is registered at the OSGi Framework under one or more interfaces. The service object should implement the interfaces under which it is registered.

Figure 2.2: Bundle lifecycle
something that is checked by the framework during service registration. Once a
bundle has registered a service object under an interface, the associated service
can be accessed by other bundles using the interface name and its methods can
be accessed by the way of its service interface. A bundle registers a service object
with the Framework by calling one of the BundleContext.registerService
methods on its BundleContext object. The registerService methods al-
low a bundle to register a service object under one or more interfaces and
an optional Dictionary object, which contains the properties of the service
as a collection of key/value pairs. Once the service is successfully registered, a
ServiceRegistration object is returned to the caller. This object is needed
in order to change the service’s properties after it has been registered. The
Framework permits bundles to register and unregister service objects dynami-
cally. Therefore, a bundle is permitted to register service objects from the time
its BundleActivator.start method is called until its BundleActivator.stop
method is called and returned.
In order to use a service object and call its methods, a bundle must first obtain
a ServiceReference object. This object is obtained by querying the register
for a service that implements a certain service interface. The query can be
more fine-granular, by providing a search filter which should be satisfied by the
services returned by the registry. In order to use the service, the bundle should
retrieve the service object from the ServiceReference object.

2.3 The Virtual OSGi Framework

The OSGi model was extended to distributed systems during previous work of
the Systems Group [21]. R-OSGi provides a seamless and non-invasive middle-
ware for accessing remote services in OSGi, by using techniques such as service
discovery and smart proxies. With R-OSGi, the service registries of remote
OSGi frameworks exchange information. When there is a demand for an ex-
ternal, discovered service, it is imported into the local service registry. This
is done with the transmission of the service interface from one registry to the
other. A proxy is created on the fly on the framework that requests the remote
service and it is registered under the received service interface. The proxy looks
equivalent to a service object to the client. Apart from that, R-OSGi handles
the dependencies that might arise from the use of a remote service too.
However, remote services and local services still have to be treated differently
before proxies are registered—unavoidable when not changing the framework
itself. In the current work on the Virtual OSGi Framework, this distinction
is eliminated by adding a thin layer on top of the isolated frameworks that
implements some functionality in a virtualization and abstraction layer and
maps other directly to the underlying physical framework. The Virtual OSGi
Framework contains a network full of machines running an OSGi Framework.
There is no distinction between local and remote services anymore. The services
installed on any machine in the Virtual OSGi Framework can be accessed from
anywhere in the network. The Virtual Framework runs as a bundle on the
host framework. Virtual bundles are installed on the host framework and are
started on the virtual framework. During installation of a bundle, the bundle is
installed on the host framework but a VirtualBundle is passed back, instead of
the host framework’s Bundle implementation. The bundle is started by getting
its Activator from the host framework, but called with `VirtualBundleContext`, which handles the virtual state of the bundle within the virtual framework. As mentioned before, there is no distinction between local and remote registry anymore. The centralized registries of the host OSGi frameworks are replaced by one distributed registry. The distributed registry is designed as an overlay network and is based on DHT properties and mainly Chord [1].

![Figure 2.3: Distributed Registry](image)

With the use of the unified service registry, the number of services installed on one framework (the virtual one) can be very high. The load on the services can also be very high, as the clients can be located on any of the nodes of the overlay network. This brings the need for an autonomous controller, which will control where to place bundles, when and which to replicate and where remote service links should be used. This approach facilitates building highly-available and fault-tolerant OSGi applications.

The motivation for this thesis becomes now apparent. This dynamic group of machines that appear as a single OSGi framework should be able to tolerate failures and balance the load. Therefore, a fluid replication mechanism is needed inside the Virtual OSGi framework. This mechanism will place a replica of a service wherever it is needed. A replica of a service is not just a plain copy. An invocation of a service method should be able to be served by any replica on the framework. Consequently, each replica should preserve the service’s internal state, which is contained in the service’s fields. Thereby, state changes inside service replicas should be captured and the updates should be propagated to every replica in the virtual framework. This is achieved with the help of coordinator nodes, which ensure consistency among all service replicas. The goal of the replication mechanism is to run with every OSGi service. As a result,
the replication mechanisms should be transparent requiring no changes in the applications. The clustering behavior is injected into the applications at load time using bytecode instrumentation. As mentioned above, the whole network of OSGi-capable nodes appears as one virtual framework having one large distributed registry. This means that the interconnected JVMs on these nodes should operate as one clustered JVM (with respect to the services). Thereby, the replication mechanism should support distributed locking and distributed thread coordination among the threads running on the JVMs of the nodes of the virtual framework. The replication mechanism can be used also for service migration, which can be regarded as temporal replication.
Chapter 3

State capturing and Bytecode engineering

3.1 Introduction - What is state?

A correct and concrete replication mechanism implementation requires preserving the same internal state across all of the service’s replicas. However, it is not straightforward to define what state is and how it can be modified. In Java and other object-oriented languages all data is organized in classes and everything is an object. A class is a collection of data, stored in named fields, and code, organized into named methods, that operates on this data. The fields and methods are called *members* of a class. Methods can additionally define and use local variables. However, local variables are only valid within the scope of a single method invocation and therefore in general do not amount to state. The data that can be regarded as state is the data contained in the fields of an object. Fields’ values can be primitive types or references to objects. In cases where a field’s value is a reference to an object, this object can contain again references to other objects in its field values. Therefore, if we tried to quantify the state of an object \( O \), we would have to follow all the paths of a big object graph, where vertices represent objects and edges represent references to objects, as shown in figure 3.1 (the grey color represents service state).

![Object Graph](image)
Every vertex in this graph that is reachable from the vertex representing object \(O\), following any possible path, is part of \(O\)'s state. In other words, every object that is accessible by reference from object \(O\) is part of \(O\)'s state. Therefore, any change of state on any of the grey vertices of the figure should be captured. A state change on a grey vertex can be originated by any other vertex in the graph that can reach a grey node through a graph's path. The complexity of state management becomes apparent, if one tries to think of the possible ways with which a path with an arbitrary start vertex and a grey end vertex can be created. Everything in Java is passed by value. This means that when passing an argument to a method or returning a value, Java copies the value and passes the copy. However 'pass by value' for reference variables has different meaning. This means that the reference to an object is copied, i.e. no new object is created, a new reference to the same object is created. In the context of the object graph introduced before, this means creating a new edge between two vertices. In other words, every local variable containing a reference to an object that is part of the state of \(O\) has automatically created an edge that has a grey vertex as an end vertex. In the general case of Java objects, such edges can be easily created if the fields are publicly accessible. But, when such an edge exists, the start vertex of the edge can easily generate a state change by changing the value of a field of a grey vertex. The object graph in this case would look like figure 3.2, where white vertices represent the local variables (not part of state).

![Figure 3.2: Object Graph with public fields](image)

Tracking all these state changes if they can be originated from any possible vertex in the above graph is really difficult, if not impossible. This could lead to a system, where everything is replicated, because the fields can be accessed by anyone or one has to explicitly state which objects are accessing a field of a grey vertex and capture all the updates in their classes' bytecode. However, in the context of OSGi services the picture is different. As mentioned in the previous chapter, each OSGi service is implemented as a service object. This does not mean that the object graph in this case contains only one vertex. Of course, the service object can reference other objects through its fields and
these objects become grey directly in the object graph. However, we now know that the object graph has the service object as a starting point. The objects outside the service’s boundaries can access the service’s state only through the service’s interfaces, i.e. only through method invocations. Interfaces cannot have fields, consequently OSGi service fields are not accessible from outside the service boundaries. This is illustrated in the object graph of figure 3.3.

![Figure 3.3: OSGi service object graph](image)

As illustrated in figure 3.3, the service is wrapped inside its interface. The clients of the service are accessing the service by calling methods of the service with the use of the interface. However, the methods are part of the service and they are known to be a 'grey' part of the graph and all the updates occurring inside these methods will be captured by the injected instructions into the service’s bytecode. No explicit statement of the places where updates can happen is needed. The boundaries of the stateful part of the graph are known and are the
same with the service’s boundaries. The white vertices of the graph can create an edge leading to a grey vertex in two possible ways:

1. When a client of a service calls the `getService` method on its context, a reference to the service object is returned. This corresponds to the creation of an edge with the client’s vertex as start vertex and the service object as end vertex. But the service object will have its bytecode instrumented and it can be accessed only via its methods. Hence, all the updates will be captured, since they will occur inside the service’s boundaries.

2. When a client of the service invokes one of its method and this method returns a reference to an object that is part of the service’s state, an edge between the client’s vertex and a service’s grey vertex is created. This means that this object can be accessed outside the service’s boundaries from now on. This is a special case that is handled by system and needs special instrumentation in that object’s class bytecode. This object will again be accessed only via its interface, so this will be the only edge created between a white and a grey vertex. With the special instrumentation done, all updates on this object will be captured, too.

### 3.2 Root objects and state definition

As described in the previous section, the objects that need to adopt clustering behavior are the objects that can be accessed by reference from the service object and they are forming a shared or clustered object graph. When an object suddenly becomes unreachable from the service object, they lose their clustering behavior. Unlike related work [25], no explicit statement of root objects of shared object graphs is needed in configuration files. Thanks to the OSGi context, the root object of a shared object graph is always the service object. From now on, terms root object and service object will be used interchangeably. Consequently, any assignment to a field that is inside the shared object graph is a state change and should be reflected at the replicas of the service. Assignments to fields and writes to fields are also two terms that will be used interchangeably.

- **Root Object**
  First of all, if a write on a field occurs inside a root object, it is surely a state change. Every field of the root object is part of the state and clearly this update should be captured. If there is a write on a field of an object that is passed as an argument, then this can be no state change under any circumstances, since the methods of a service’s root object are accessed by the ‘outside world’, i.e. outside the service’s boundaries, and every argument that is passed cannot be something that is part of the service’s state. Eventually, if a write on a field of an object that is referenced by a local variable occurs, then this will lead to a state change if the local variable references an object referenced by a root object’s field or another object accessible by reference from a root object.
3.3 Symbolic code analysis

3.3.1 Overview

In order to track all the updates that can happen inside the service’s code, it is necessary to track what exactly is pushed on the stack and what is stored in local variables and fields. The contents of the operand stack and the local variables are changing continuously during a program execution. Furthermore, their content can depend on execution of other methods. As shown above, the updates can happen on a field or a field referenced by a local variable. With the argument passing from one method to another the content of a variable is highly dynamic and can be changing from being part of the state or not. Therefore, a technique of tracing the data flow during the code execution is required. One possible way of doing so is through symbolic code analysis \[15\]. During symbolic code analysis, symbolic execution of the programs is performed. Instead of supplying the normal input to the program, symbols representing arbitrary values are supplied. Below follows an exploration of the basic concepts and constructs used in the symbolic code analysis mechanism.

3.3.2 Symbolic analyzer

Code analysis is a very large topic and many algorithms exist for analyzing code. Two are the most important types of code analysis.

- **Data flow** analysis is a technique for gathering information about the possible set of values calculated at various points in a computer program. During symbolic execution, the possible set of values is represented in an
abstract way with the use of symbols. With the use of symbols, useful information about state changes during the execution can be derived.

- A control flow analysis consists in computing the control flow graph of a method and in performing analysis on this graph. The control flow graph is a graph whose nodes are instructions and whose oriented edges connect two instructions \( i \rightarrow j \), if \( j \) can be executed just after \( i \). Control flow analysis is used for bytecode instrumentation to enable thread state capturing (See chapter 5).

Data flow analysis

Data flow analysis can be performed in two ways.

- **Forward** analysis
  For each instruction, the state of the execution frame after this instruction is computed, from the state before its execution.

- **Backward** analysis
  For each instruction, the state of the execution frame before this instruction is computed from the state after its execution.

For the symbolic analysis of the code a forward data flow analysis technique was used with the help of the Analyzer class of ASM [8]. A forward data flow analysis simulates the execution of each bytecode instruction of a method on its execution stack. It pops values from the stack, combines them and pushes the result back on the stack. Forward data flow analysis seems similar to what a Java intrepeter does, but it has some significant differences. First of all, a Java intrepreter follows only one control flow path while a data flow analyzer should follow all possible control flow paths. By following all possible control flow paths, the forward data flow analyzer cannot say what value exactly will be pushed on the stack. It should compute a set of possible values that can be on the stack. This means that the analyzer will simulate the execution of an instruction \( i \) by finding the set of all possible results of \( i \) for all combinations of values in its operand value sets. This is done by pushing objects on the simulated stack that represent a specific set of values with a specific data type.

As explained above, knowing just the types of the values that are on the stack was not enough. Some more information about the values on the stack was needed. So, our own implementation of an Analyzer was created. The SymbolicAnalyzer performs forward data flow analysis for a method and produces a method descriptor (see following section). The SymbolicAnalyzer encapsulates a SymbolicInterpreter which consumes events produced by ASM’s Analyzer according to the instruction simulated. The events produced by the Analyzer are

- **copyOperation**: bytecode instruction that moves a value on the stack or to or from local variables.
- **unaryOperation**: bytecode instruction with a single argument.
- **binaryOperation**: bytecode instruction with two arguments.
- **ternaryOperation**: bytecode instruction with three arguments.
3.3 Symbolic code analysis

- *naryOperation*: bytecode instruction with variable number of arguments.

When producing each of these events, the *Analyzer* passes the instruction that is interpreted and the arguments to the *SymbolicInterpreter*. The *SymbolicInterpreter* then interprets this instruction and returns the *symbol* to be pushed on the stack, according to the bytecode instruction interpreted and the arguments of this instruction (symbols). Hence, it is clear now that by intercepting the simulation of the bytecode interpretations and the injection of symbols, the exact behavior of a method can be memorized. The operations needed to be memorized will be better understood, if the symbols created and their meaning is introduced.

**Symbols**

As mentioned before, what is needed to be captured are the updates of fields of objects that are part of the state (i.e. part of the service object’s object graph). So, the most important instruction interpreted is the *PUTFIELD* instruction. However, the arguments of the *PUTFIELD* operation need to be parsed, that’s the reason why all of the instructions are interpreted. The arguments of the *PUTFIELD* operations are encapsulated in the following symbols. Each symbol contains information for the type of value it represents.

- **ThisSymbol**
  As the name implies, *ThisSymbol* represents the *this* reference for an object.

- **LocalVariableSymbol**
  It represents a local variable that is created in a method. It simulates the way JVM is operating on local variables, as shown in figure 3.4.

- **ArgumentSymbol**
  It represents the content of the local variable when it is passed as an argument to the method.

- **FieldSymbol**
  It represents a field value. It is pushed on the simulated stack either when interpreting a *GETFIELD* instruction, or when dereferencing the content of a local variable and the local variable has been assigned a field value.

- **MethodCallSymbol**
  It represents a method invocation. This symbol is used to track the nested method invocation, so as to track the updates that this method does.

- **StaticFieldSymbol**
  Similar to the *FieldSymbol*, but it represents a class field and not an instance field.

- **MonitorSymbol**
  It represents a cluster-wide synchronized block.

- **PhiSymbol**
  It represents the union of the possible return values of a method, when it can return a value from more than one branches of the control flow.
- **ArrayFieldSymbol**
  It represents an element of an array that may amount to state.

- **ExternalFieldSymbol**
  It represents a field of an external object that may amount to state. External object represents an instance of a JDK, whose code cannot be instrumented.

![Local variable symbol](attachment:image.png)

**Figure 3.4: Local variable symbol**

**Method descriptors**
As was mentioned above, the analysis is done in order to see if the method contains any state changes. If yes, bytecode instructions are injected that will give clustering behavior to the method. The method analysis implies that the execution of the method is simulated. This means that a method analysis can lead to a huge invocation graph, since the execution of the nested invocations has to be simulated too. It is obvious that the analysis can create great time overhead. For this reason, the method analysis needs to be optimized as much as possible. The key idea to achieve the optimization of the method analysis is that the behavior of a method will always be the same, irregardless of the caller method. The only difference is that the writes that occur in the method can sometimes cause a state change and sometimes not depending on the context on which the method is invoked. This means that the possible updates of a method have to be modelled in a structure that can be cached and reused by caller methods by giving their call context and getting the updates of this method according to the call context. This structure is the MethodDescriptor. A MethodDescriptor is created once the analysis of method is complete and it contains the method’s writes and its return value.

**Method descriptor resolution**
The processing of the symbols does not stop here. What has been achieved till now is to create descriptors for the methods which will contain only FieldSymbols and to analyze every nested method that appears on the call hierarchy. However, this can create a high overhead. Imagine analyzing a method that is being called very often every time one invocation of it is encountered. This would mean simulating the interpretation of its bytecode instructions each of these times. And this would have no meaning as the behavior of its code will be the same for all of these invocations. If the
method writes on a field, this write will happen in every invocation. The only thing that may change is the field that it writes which may depend on the call context. For the above reasons two optimization techniques are being used.

1. **Caching of method descriptors**
   When a method is analyzed its newly created descriptor is being cached in a global location. When another method is invoking this method, it will not have to analyze this method (and all the descendant methods that are invoked by it). It will fetch its descriptor instead which will be containing its writes (and all the writes of the descendant methods invoked by it) in a reusable format.

2. **Descriptor resolution**
   When a method descriptor is reused, the method’s writes have to be resolved because they depend on the call context. Hence, when a method invocation is added as a write, its descriptor is fetched from the cached location (or created if not present) and its writes are being inspected. The symbol resolution of the writes needs the symbols of the arguments with which the method was called.

### 3.4 Replication constructs

Before describing the process of injecting the clustering behavior by instrumenting the bytecode of a class during load time the basic constructs used during instrumentation to capture state changes have to be described.

#### 3.4.1 Transactions

The method invocations done on a service interface are modelled with transactions. A transaction collects all the updates done during a method’s invocation time and is propagated to the other replicas, as is described in the following chapter. A transaction is initiated from each method of the root object and is passed as an argument to the instrumented nested method invocations (and all the descendants in the call hierarchy). The transaction boundaries are usually the beginning and the end of a root method invocation. However, when a synchronized block is encountered the current transaction is terminated (and propagates its updates) and a new transaction is initiated for the synchronized block. This is required, as the time required to enter a synchronized block cannot be specified and the delay of the updates propagation because of a thread waiting to acquire a lock (local or cluster-wide) can lead the replicas to an inconsistent state. It was mentioned that the transaction is passed from the service object to the nested method invocations. This means that the signature of a nested method should be changed to contain the transaction argument. Hence, if a nested method contains updates of fields that amount to state, a new method is created with the same behavior but with an additional argument—the currently executing transaction. When a method will lead to state change with a certain call context (inferred from the symbolic code analysis) it will have two versions, the original one and the one with the additional transaction argument. Through the analysis, each method knows if each nested invocation it makes will lead to state change or not. In the first case, it will call the instrumented version of the
method, by passing as an additional argument the transaction it is executing and in the second case it will call the original method.

As mentioned above, the writes done by a nested method can be a stage change or not depending on the call context. The writes a method does can be on instance or static fields of the instance on which the method invocation is done, or on fields of the objects referenced by arguments that are passed to the method. A method will be called with the transaction argument only if at least one of its writes is a state change. At this point a small problem arises. The calling method knows exactly which of the writes during this nested method invocation are state changes and which are not. The called method has its code instrumented. But this method can be called various times and some of this times the arguments can amount to state or not and the same for the instance’s fields. So, there is no way for the called method to know which updates to store in the transaction and which not, while it is being executed. Therefore, the calling method should pass the call context to the called method and the called method should check during its execution if the call context amounts to state and either store the updates or not. However, efficiency if of great importance to minimize additional overhead for the instrumented method. The call context is pushed into a stack that is held by the transaction, just like JVM pushes the arguments of a method call into a newly created frame in the thread’s stack. The transaction creates new frames in its stack as it is being propagated from one method to another and pushes the context of each method. The call context is pushed into the transaction’s stack in the form of hierarchical identifiers, starting with this which identifies the service object and adding the names of the fields, just like the reverse domain naming schema used in the Java language. If the context is not stateful, null is pushed in the transaction’s stack. For example in case the following method in the root object

```java
public void methodCall(){
    field.nestedCall(new Object());
}
```

which calls the nestedCall method that does some updates with this call context the above code will be instrumented to the following

```java
public void methodCall()
```
public void methodCall(){
    Transaction t = new Transaction();
    t.pushContext("this.field", null);
    field.nestedCall(new Object(), t);
    sendUpdates(t);
}

and the stack of the transaction will look like in figure 3.6.

![Figure 3.6: Transaction’s context stack](image)

In this way the `nestedCall` method will know that during this method invocation, if it writes on a field of the instance on which it is called, it should add it to the transaction’s updates and if it writes on a field of the object referenced by the argument, it should not add it to the transaction’s updates. The same method could be invoked with an argument that amounts to state from another place in the code. This invocation will push the identifier of the object that is passed as an argument on the transaction’s context stack and the called method will know that any write on any field of the object referenced by the argument is a state change and should be added in the transaction. The check required in the nested method call is just a check if an element of the transaction’s context stack is `null`. The context stack is essentially an array, so the overhead of this check is minor. In the above example, if the `nestedCall` method calls another method, then it will push on the context stack the call context, which will be renamed appropriately to follow the hierarchy with respect to the root object. For example if `nestedCall` calls a method with no arguments on a field `nestedField`, the stack will have the form illustrated in figure 3.7 during the method invocation.

![Figure 3.7: Transaction’s context stack](image)

As depicted in figure 3.7 a new simulated frame has been created containing the
identifier of the field on which the method is called. The transaction offers two useful primitives to the methods participating in it:

- **void add(TransactionEntry entry)** adds an update to the currently executing transaction.
- **String[] getContext()** retrieves the call context array.

### 3.4.2 Updates

The updates are constructed using the **Command** design pattern [9], which is an object behavioral pattern to achieve complete decoupling between sender and receiver. (A sender is an object that invokes an operation, and a receiver is an object that receives the request to execute a certain operation. With decoupling, the sender has no knowledge of the Receiver’s interface.) The term request here refers to the command that is to be executed. The **Command** pattern also allows variation on when and how a request is fulfilled. Therefore, the **Command** pattern provides flexibility as well as extensibility. In our case, the **Command** pattern helps in decoupling the execution of an update at a replica from the entity that initiates the execution of an update. There are two implementations of the TransactionEntry interface:

#### FieldUpdate

This transaction entry represents updates that assign a new value to a field. The **FieldUpdate** holds the following information about the update.

- **objectID** this is the identifier for the field updated in Java notation.
- **value** this is the value that is assigned to the field. The value of the update will be the object if this object value has not been sent over the network, a reference identifier if this object has already been sent over the network, or a **FieldValue** object if what is assigned to the field is the value of another stateful field. **FieldValue** holds the unique identifier of that field in Java notation.
- **arrayIndex** this is the index of the element updated into the array, if the field updated is an array.
- **refId** this a reference id. A unique reference id is assigned to every object serialized and sent over the network. This reference id prevents from serializing the same object more than one times when it is sent more than one times over the network. The reference id helps also in preserving the identities of the objects. When an object is serialized, sent to a replica and instantiated on the heap of the target location, then every change at this object will be done on the object being located on that replica’s heap. There will be only one copy of that object in each replica’s heap. More about reference identifiers will be described in the next chapter.

#### MethodInvocation

This transaction entry represents updates that execute an external method, i.e. a JDK method or a method outside the scope of the service. **MethodInvocation** holds the following information about the update.
3.5 Injection of Clustering Behavior–Technical Description

- **signature** this is the signature of the method being invoked.

- **refIds** these are the reference identifiers assigned to the arguments if it is the first time that they are serialized and sent over the network.

- **args** these are the arguments of the method invocation. The arguments, like the value of a FieldUpdate can be objects, if it is the first time that they are serialized and sent over the network, reference identifiers if they have already been serialized before or FieldValues if some of the arguments are values of stateful fields. At least one of the arguments will be a FieldValue, otherwise the method invocation could not be stateful.

3.4.3 Referenced objects

Another interesting construct used during instrumentation are the ReferencedObjects. These are the instrumentation constructs that correspond to the ComplexSymbols used during symbolic code analysis. They are used to reference objects that amount to state because they are elements of an array or part of the state of fields that are JDK objects (and their bytecode cannot be instrumented). The ReferencedObjects are created because these objects cannot be identified by using field identifiers in Java notation as it is done with other fields. These objects can only become state by the use of a field assignment or a method invocation. So, these objects will have surely acquired a unique reference id, once they become state. As a result, these objects are referenced uniquely with the use of their reference ids. The information held by the ReferencedObjects is

- **refId** this is the unique reference identifier for this object.

- **fieldName** this the name of the field that contains this object as part of its state, i.e. the name of the array or the name of the JDK field (Map, List etc.).

3.5 Injection of Clustering Behavior–Technical Description

Once the analysis is completed, all the required information for injecting the clustering behavior has been collected and the bytecode instrumentation can take place. The corresponding sequence diagram is shown in the figure 3.8. All the classes that are analyzed are loaded using a ch.ethz.iks.instrumentation.impl.ClassAdapter. ClassAdapter which constructs the elements of a class by the use of a ClassReader.

Each class is loaded once and its ClassAdapter is cached in a global location. The ClassAdapter adds two new fields in each class that is loaded, a ReplicaNode and a Scheduler located in the package ch.ethz.iks.comm. The ReplicaNode is used for communicating with the replicas coordinator and the Scheduler is used for executing the updates asynchronously. Both of these constructs will be described in the following chapter. ClassAdapter injects setter/getter methods for these two fields too. Each ClassAdapter encloses a ClassWriter which writes the bytes of the class and after that the class is ready for loading. Before writing the bytes of a class, the ClassAdapter
calls `instrument` on each of the class’s methods. A new `MethodAdapter`, located in the `ch.ethz.iks.instrumentation.impl` package is created when the `visitMethod` method of `ClassAdapter` is called by the `ClassReader` that reads each class. The `MethodAdapter` checks the instrumentation flags to check which type of instrumentation technique should be used. The available flags are

- **isThreadInstr** this flag indicates that this method is accessed by a service’s internal thread in a context not causing any stateful updates. This means that the original method should be instrumented for thread state capturing as described in the chapter 5.

- **isReplInstr** this flag indicates that the method is accessed in a context that causes some updates, but it should not be instrumented for thread state capturing. In this case the original method is written as it is and a new method is created with the additional transaction argument (if it is not a method of the root object).

- **isThreadAndReplInstr** this flag indicates that the method is accessed in a context that causes updates and it is accessed by a service’s internal thread in such a context too. This means that the method version with the additional transaction argument should be instrumented for updates capturing and thread state capturing. The `isThreadInstr` specifies what should be done with the original method (if it should be written as it is or instrumented for thread state capturing) and the other two flags specify what should be done with the method version with the additional argument. In the `isReplInstr` case this method is instrumented only for updates capturing and in the `isThreadAndReplInstr` it is instrumented for both updates capturing and thread state capturing.

- **isComplexObject** this flag indicates a special case. This case occurs when an object that is part of the state i.e. it is accessible by reference from the service object is a return value of one the root object’s methods. In this case, the clustering behavior of this object should be retained. But the object has left the service’s boundaries, so the instrumented methods for updates capturing, i.e. with the additional transaction argument, will not get invoked anymore. For this reason, the original methods of this class should be instrumented in a special way that will capture the updates...
happening to objects leaving the service's boundaries. This technique will be described in the following section.

- **returnsComplexSymbol** this flag indicates that this root method returns something that amounts to state. This means that the corresponding class will be treated specially as mentioned above but some fields of the object returned should be initialized to indicate that this object has left the service's boundaries.

The instrumentation for thread state capturing is described in chapter 5. In the case of instrumentation for updates capturing, two cases are distinguished.

### 3.5.1 Service object method

When a method of the service object is instrumented, a new transaction object is created in the beginning of each method. This object is stored in a local variable which is subsequently used to reference the transaction in the whole method body. Each transaction registers the fields that it will update. These fields are registered with their unique identifier in Java notation during the construction of the transaction. This registration locks these fields for use in this transaction. If an update comes from another replica during the period that these fields are locked, there should be a conflict for some field, as two replicas have updated the same field concurrently. The description of such a case and its resolution is described in the following chapter. The fields that will be updated during a transaction are determined during the symbolic code analysis which has followed all the call hierarchy and has determined exactly which fields will be updated. The locking of these fields is done with the call of the \texttt{BOT} (Begin of transaction) primitive of the \texttt{ReplicaNode} field that each replicated object has. This primitive triggers the beginning of a transaction and locks the appropriate fields. The unlocking of these fields is done when the transaction ends, when the \texttt{EOT} (End of transaction) primitive will be called.

Once the transaction has begun, the updates done inside only this method and not the whole hierarchy have been stored during the symbolic code analysis. Now they are parsed to insert the capturing code. There are four possible cases:

1. **Field update**

   The field updates are classified into three categories. The primitive value and reference type fields, array type fields and complex type fields, which are updates on fields of objects that are array elements or part of a JDK’s object. In the case of primitive value and reference type fields, the actual \texttt{PUTFIELD} instructions are intercepted with the injection of an instruction that stores the newly assigned value into a temporal local variable. If the value is a primitive type, it is boxed into the corresponding complex type. If the assigned value corresponds to another field that amounts to state, then a new \texttt{FieldValue} is created instead of capturing the actual value. This will prevent the case of serializing the whole object, because it is not needed, as the other replicas will have the same value in the field, whose value is assigned. The \texttt{FieldValue} contains the unique identifier of the field. Whatever it is the value assigned, a new \texttt{FieldUpdate} is created which contains the unique identifier of the field to be updated, and the value assigned, which as mentioned above can be a \texttt{FieldValue} or the
actual value that is assigned. As we are inside a service object method, a FieldValue is assigned only if the value that is assigned is a value of a field that has this as its owner, i.e., it is reachable by reference from the service object. This can be determined by the symbols created during the symbolic code analysis. The FieldUpdate is added to the currently executing transaction, which can be the transaction of the method, or the transaction of a synchronized block, if the update is inside one. The second case of a FieldUpdate is a write on an array that is a field amounting to state. The difference in this case is that the value of the update has to be stored together with the index inside the array where this value is stored. The assignments on an array element have always a pattern similar to the one shown below, so the value assigned and then the index inside the array are popped from the stack and stored into local variables.

```classfile
\// PUSH reference to array
BIPUSH index
\// PUSH value
AASTORE
```

The FieldUpdate created in these cases has the isArray flag set and carries the array index inside it. The last type of updates on fields are the updates on a field of an array’s element and the updates on a field of an external stateful object (JDK object or object outside the service’s boundaries). This case is a little more complicated, since this object’s class bytecode cannot be instrumented. Hence, in this case, the instruction in which a reference to this object is pushed on the method’s stack frame has to be tracked. First of all, the value that is assigned to the field needs to be saved in the same way that it was saved for the other cases. Subsequently, the instruction that loaded the object on the stack is tracked. This instruction is identified with the help of a symbol created during the symbolic code analysis. This symbol holds the instruction that pushes the object on the stack. Then, this instruction is intercepted and a reference to this object is saved in a local variable. This object is an element of an array that amounts to state or part of a JDK object’s state. This means that it has to have been assigned a value before. Consequently, this object should have been serialized and assigned a unique reference identifier. This reference id is retrieved with the help of the ReplicaNode’s getRefId primitive. Even if a reference identifier has not been assigned to this object, the ReplicaNode will assign the next available reference identifier to the object during the call to getRefId and it will propagate it to all replicas of this service. Then, a new FieldUpdate will be instantiated containing a ReferencedObject as the object identifier. This ReferencedObject will be identified from the other replicas using the reference identifier it contains and the identifier for the field of this object that is updated. Using the unique reference identifier, each replica will locate the appropriate object instantiated on the local heap and do the assignment to its field. The value of the FieldUpdate can again be a FieldValue or a reference to the value that is being assigned.

2. Method invocation

When there is a nested method invocation inside the updates gathered
from the symbolic analysis, this means that the nested invocation causes some updates with this specific call context. This means that the full transaction context has to be provided to this method invocation. For this reason, the `INVOKE` instruction is modified to contain the modified signature of the method with the additional transaction argument. The currently executing transaction, which can be the transaction created in the beginning of the root object’s method, or one created when entering a synchronized block is pushed on the stack just before the method invocation as the last argument to the method. Furthermore, right before passing the transaction as an argument, the call context is pushed on the transaction’s stack, by creating a new stack frame for the called method. The call context is popped just after the method invocation. Pushing the call context is performed through the `MethodCallSymbol` created for this nested method invocation during the symbolic code analysis. This symbol contains the arguments of the method invocation in a symbolic form. While injecting the bytecode instructions that push the context on the transaction’s stack, the symbols that are the arguments of the `MethodCallSymbol` are checked. If an argument is a `FieldSymbol` it is pushed on the transaction stack, only if its owner is a `ThisSymbol`. Because of the fact that we are in a service object’s method, a field can only amount to state, i.e. be reachable by reference from the service object, only if its owner is the `ThisSymbol`, i.e. the service object. If the owner of the field is an `ArgumentSymbol`, this field can under no circumstances amount to state, as the arguments passed to the service’s method are stateless, because they are coming from outside the service interface’s boundaries. Hence, if the field amounts to state, its identifier is pushed on the transaction’s stack. If an argument is the `ThisSymbol`, it always amounts to state and the `this` identifier is pushed on the transaction’s stack. If one of the arguments is a field of an array’s element or a field of an external object, a new `ReferencedObject` is created for it, by retrieving its unique reference identifier as mentioned before, and it is pushed on the transaction’s stack. In all other cases, the object referenced by the argument does not amount to state, and `null` is pushed on the transaction’s stack. Finally, the `ReplicaNode` and `Scheduler` used will be passed to the transaction, to be used by the nested method invocations.

3. **External method invocation**

The next category of updates is a method invocation on an external object, i.e. a JDK object or an object that crosses the boundaries of the service. In this case the same method invocation should be executed on all replicas of the service, as the method’s bytecode cannot be instrumented and the method’s updates captured in a more fine granular way. Bytecode instructions are injected that create a new array with length equal to the number of arguments of the method. This array is a `java.lang.Object` array to store every possible argument type. Then all the arguments of the method are stored in this array. These bytecode instructions are injected just before the `INVOKE` bytecode instruction, as the Java stack at this point will contain the arguments in the top positions, in the order they are pushed on the stack to prepare the method invocation. If any of the arguments is a primitive type it will be boxed into the corresponding reference type. If
any of the arguments happens to amount to state (at least one of them will amount to state, otherwise the whole method invocation cannot cause a state change), an appropriate FieldValue will be created, which will contain the unique identifier for this field. As before, a field can be stateful if the corresponding symbol created by the SymbolicInterpreter has ThisSymbol as its owner. Once the arguments of the method have been saved in the appropriate format, a new MethodInvocation which contains the method's signature and its arguments is added at the currently executing transaction.

4. MONITORENTER, MONITOREXIT instructions

In cases of synchronized blocks on monitor of an object that amounts to state, the synchronized block will have to be modified to a cluster-wide synchronized block. So, like the usual case in a single JVM where only one thread of the JVM is allowed to execute instructions inside a synchronized block at a certain time, now only one thread across all the clustered JVMs should be allowed to execute instructions inside a cluster-wide synchronized block. The synchronized blocks are tracked during symbolic code analysis when MONITORENTER and MONITOREXIT instructions are met. When a MONITORENTER instruction is detected, that is executed on a monitor of an object that amounts to state, i.e. a FieldSymbol that has ThisSymbol as owner, a new MonitorSymbol is created. These symbols are registered in the SymbolicInterpreter in order to keep track of the updates that are done inside synchronized blocks. This is required because the synchronized blocks are executed inside separate transactions and the fields that each transaction should lock have to be specified, as mentioned before. The updates that are made inside a synchronized block are marked with the block’s MonitorSymbol. When a MONITOREXIT instruction is met, the corresponding MonitorSymbol is removed from the registry and is no more the active synchronized block. The MONITOR instructions on a lock of an object that amounts to state are stored to be modified during the instrumentation. When instrumenting these instructions, first of all the end of the currently executing transaction is triggered by calling ReplicaNode’s EOT for this transaction. After that, a goto instruction is added just before the MONITORENTER instruction. In this way, the local lock is not acquired and the control flow jumps to the instruction that acquires the cluster-wide lock. This is the method distMonitorEnter of ReplicaNode, which takes as an argument the unique identifier of the field whose lock is to be acquired. The functionality of this method will be described in the next chapter. After the lock acquiring code, bytecode instructions that create a new transaction are injected in the method code. This new transaction declares the fields that it will modify (inside the synchronized block) and locks them when it triggers its beginning by calling BOT. Likewise, when the bytecode transformer meets a MONITOREXIT instruction, it inserts a goto instruction just before it and transfers the control flow to the call of distMonitorExit which releases the lock having been acquired (with the use of its identifier). When the lock is released, the transaction of the synchronized block is terminated and the transaction that was executing before entering the synchronized block is resumed.

5. wait/notify instructions
The mechanism that Java uses to support synchronization is the monitor. Java’s monitor supports two kinds of thread synchronization: mutual exclusion and cooperation. Each object has its own monitor. When an object becomes replicated among many JVMs its monitor should become replicated too. Mutual exclusion is supported in the Java virtual machine via object locks. In our system, the locks become replicated by the bytecode injection described before. But in order to make the whole monitor have cluster-wide semantics, the cooperation properties of the monitor have to be enabled to have cluster-wide meaning. This is done by instrumenting the wait/notify/notifyAll methods that operate on a monitor to be replaced by the cluster-wide ones that operate on a distributed monitor. These method invocations are again tracked during the symbolic code analysis and more specifically during the symbolic interpretation of a naryOperation when the name of the method is wait, notify or notifyAll and the owner is the java.lang.Object. These instructions are stored only when the monitor on which they operate is a monitor of an object that amounts to state. The technique used to make the instructions operate on a distributed monitor is the same used for the instructions of synchronized blocks. A goto instruction is injected just before the original wait, notify or notifyAll invocation which drives the control flow to the invocation of the corresponding method with cluster-wide semantics. These methods (distWait, distNotify and distNotifyAll) declare the monitor on which they operate by using the unique identifier of the stateful field. When a wait method invocation is encountered, this implies that we are in a synchronized block. The enclosing synchronized block is located, in order to terminate the currently executing transaction, as the wait time can be long and lead the replicas to an inconsistent state. Apart from that, the fields accessed by that transaction should be unlocked while waiting. After that, the distributed version of wait, i.e. distWait is invoked, with the corresponding identifier for the stateful field, as determined by the symbol created during symbolic code analysis. The enclosing transaction is resumed when distWait returns. The internal implementation of distWait is described in the following chapter. In cases of distNotify and distNotifyAll nothing should be done with the enclosing transaction and the only thing required is to drive the control flow to the distributed versions of the notify/notifyAll methods.

3.5.2 Other methods

The first step during instrumentation of a non-service-object method is to create the new signature for the method. A new local variable is created which can hold the transaction context. The transaction context is added to the end of the arguments of the method. If the method uses some local variables, which is the most likely case, this index will be in use from the existing code. In this case the conflicting variable is adjusted to use the next available index in the constant pool by modifying all the instructions that use it. After that, the call context of the method is extracted from the current transaction and stored in a local variable. The ReplicaNode used is also extracted from the transaction and stored in the appropriate field.

The updates happening during a transaction in a method of an object that is
not the service object are handled in a similar way to how they are handled in the service object but with some slight differences. The differences are described below according to the category of the update.

1. **FieldUpdate**
   The difference in this case is that it has to be checked whether the field updated amounts to state. So, the appropriate index in the call context array is checked if it is `null` or not, depending on whether an instance field or a field of an object that is passed as an argument is being updated. If the corresponding call context is not `null`, the value of the update is saved as in the case of the service object case and a **FieldUpdate** is added in the currently executing transaction, which is either the transaction passed as an argument to the method or a newly created transaction for a synchronized block inside this method. In cases of updates on array element, the only difference is again the check if the array is stateful, by checking again the appropriate call context. The same differences hold for the updates on a field of an array’s element and the updates on a field of an external object (JDK object or object outside the service’s boundaries) that amounts to state.

2. **MethodInvocation**
   The handling of the nested method invocations is almost the same for methods of objects other than the service object, concerning the bytecode instrumentation. The only difference from the bytecode instrumentation view, is that identifiers for fields that are owned by an **ArgumentSymbol** are pushed too. In other words, everything that can amount to state is pushed on the transaction’s call context. But this context can amount to state or not, depending on the call context on which this non-root method has been called on. The filtering of whether the arguments of the nested method invocation amount to state or not is done inside the **pushContext** primitive of the currently executing transaction. This method creates a new virtual stack frame for the nested method invocation while pushing the context, as mentioned before. But when it creates a new stack frame and the stack is not empty (i.e. we have a nested method invocation inside an already nested method invocation) it checks the context on which the calling method has been called, in order to determine the context of the called method. An illustration of this filtering process will make things clearer. If there is a method invocation inside the service object like the following one.

```java
public void rootMethod(){
    rootField.nestedMethod();
}
```

and the body of the nested method is like below

```java
public void nestedMethod(){
    nestedField.nestedNestedMethod();
}
```

the transaction’s virtual stack will have the snapshot depicted in figure 3.9 after pushing the context for the **nestedMethod**.
When `nestedMethod` is pushing context on the transaction’s virtual stack it pushes the identifier `this.nestedField`. Then the transaction makes the filtering by checking the call context on which `nestedMethod` has been called. This results in the transaction stack snapshot of figure 3.10.

As illustrated from figure 3.10, the transaction renames the identifier of `nestedField` from `this.nestedField` to `this.rootField.nestedField` to express the hierarchy with respect to the root object. Another example is the following calling hierarchy.

```java
public void rootMethod()
{
    rootField.nestedMethod(new ArgObject());
}
```

This leads to the transaction stack’s snapshot illustrated in figure 3.11, since the object referenced by the argument is not state.

If the body of `nestedMethod` was:
public void nestedMethod(ArgObject arg){
    arg.nestedNestedMethod();
}

then it pushes the identifier arg1 to the transaction’s stack. The transaction checks the calling context of nestedMethod to rename the arg1 identifier, leading to the following transaction stack’s snapshot as the object referenced by the first argument of nestedMethod does not amount to state. If an object referenced by an argument does not amount to state, no matter what the calling context is, null will be pushed on the transaction’s virtual stack. Finally, the ReplicaNode and Scheduler used will be passed to the transaction, to be used by the nested method invocations. These fields are available in the calling non-root method from the transaction passed to it from the service object as an argument.

Figure 3.12: Transaction’s stack for nestedNestedMethod

3. External method invocation
In case of an external method invocation, the writes of the external method are parsed in order to see on which object’s fields that is passed as an argument the method writes. These objects passed as arguments should be checked if they amount to state or not, with the help of the call context on which this method has been called. So, the writes of the external method are parsed and according to the owner of each FieldSymbol that it is contained in the writes the corresponding element in the calling context array is checked if it is not null. For every argument that the corresponding calling context is not null, a new FieldValue with a certain identifier derived from the symbol of this field is created and added to the transaction. The transaction renames the names of the identifiers according to its virtual stack frame, as it is also done in the case of context pushing described above.

4. MONITORENTER, MONITOREXIT instructions
The MONITORENTER and MONITOREXIT instructions inside methods of non-root objects are handled in a similar way to the methods of the service object. The main difference is again that such methods can be called with different context each time, so it is not known until runtime if the lock to be acquired belongs to an object that amounts to state or not and consequently if a local lock should be acquired or a cluster-wide one. So, like the other cases, the corresponding element in the call context array should be checked at runtime. If the context is not null, the local JVM’s MONITORENTER is skipped and distMonitorEnter is invoked for the field,
by passing its identifier (appropriately renamed with the corresponding context) and the cluster-wide lock is acquired. If the context is **null**, the local JVM’s `MONITORENTER` is executed and the local lock is acquired as it would be if the object had no clustering behavior. The previously executing transaction is terminated and a new one is created. Before, triggering the begin of the new transaction, the call context of the previously executing transaction is propagated to the new one. Similarly, when the bytecode transformer meets a `MONITOREXIT` it injects the code that will skip this instruction and call `distMonitorExit`, when needed, will terminate the transaction of the synchronized and resume the previously executing one.

5. **wait/notify instructions**
The only difference of this case with the one encountered in the service object is that again the call context has to be checked to see if the monitor used belongs to an object that amounts to state or not. If it amounts to state, the distributed versions of the methods are called and if not the original versions of the methods are invoked. The enclosing transaction is again terminated and resumed, in case of a `wait` method invocation.

**Synchronized methods**
The Java virtual machine does not use any special opcodes to invoke or return from synchronized methods. When the virtual machine resolves the symbolic reference to a method, it determines whether the method is synchronized. If so, the virtual machine acquires a lock on the enclosing instance before invoking the method. After a synchronized method completes (regardless of completing by returning or by throwing an exception) the virtual machine releases the lock. Hence, a synchronized method on an object that amounts to state has to acquire the cluster-wide lock for the duration of the method execution. In methods of the service object, a call to the `distMonitorEnter` primitive of the `ReplicaNode` is just added after creating the transaction and before calling `BOT` for it. The `distMonitorEnter` is called for this. Just before calling `EOT` when the transaction is over, the cluster-wide lock is released by making a call to the corresponding `distMonitorExit`. For non root-methods, some more processing is needed. First of all, the call context at index 0 has to be checked if it is not **null**. This means that the object on which the method is invoked amounts to state, so the cluster-wide lock for this object has to be acquired. In this case, the previously executing transaction has to be terminated because each synchronized method has its own transaction. The previously executing transaction in this case is the one passed as an argument. After acquiring the cluster-wide lock, a new transaction begins which ends after releasing this lock. When this transaction begins, the call context of the method is pushed into this transaction’s stack too and the fields updated in this method are added to the locked fields for this transaction. These fields are again known by the symbolic code analysis and they are expressed with the help of the call context. Before returning, `BOT` has to be called again for the previously executing transaction.
3.5.3 State crossing the boundary of the service interface

A special case of instrumentation is happening when part of the service’s state leaves the service interface’s boundaries. This can happen for example when a reference to a service’s field is returned by a service’s method, or when something stored into one of the service’s data structures, e.g. HashMap or ArrayList is returned by a service’s method. In these cases, a way to capture any updates happening to this object, while it is manipulated outside the service, is required. The difficulty in capturing the updates with the instrumentation mechanisms described until now, lies in the fact that the state capturing in objects other than the service object was done by passing the executing transaction as an additional argument to an instrumented version of the method which stored the updates at the executing transaction. But now that an object crosses the service’s boundaries, the instrumented methods will still be in the bytecode but they will never be called. The ‘outside world’ is not instrumented and is oblivious to the fact that the object returned by the service is replicated. After all, that was the initial goal (transparency). The user of the service should be unaware of the replicated behavior. In this special case the caller of the service should continue being unaware of the replicated behavior but at the same time this behavior should remain present. This means that the updates happening on these objects should continue being captured despite the fact that the ‘outside world’ will be invoking the original methods. An obvious consequence is that the original methods of these objects’ classes have to be instrumented, but in a way that this capturing mechanism is initiated when an object that amounts to state crosses the service’s boundaries.

This different instrumentation and capturing technique starts again from the symbolic code analysis, as expected. Once the analysis of a root method is finished, its return values are checked. If the method returns something that amounts to state in one of the possible control flow branches this method’s MethodAdapter is marked with the flag returnsComplexSymbol, which was mentioned above. The stateful object can either be a FieldSymbol with ThisSymbol as its owner, which is a field reachable by reference from the root object, or a ComplexSymbol, i.e. an element of an array that amounts to state or an external object’s field that amounts to state, or a FieldSymbol with a ComplexSymbol as its owner. In any case, the class of the object that is returned is marked for special instrumentation. The instrumentation starts by inserting three new instance fields to the class. These are three fields required for the clustered behavior. The first one is the ReplicaNode field which is responsible for the communication and coordination of the replicas. The second one is the Scheduler field which is responsible for the asynchronous execution of the updates and the third one is the ReferencedObject which is responsible for the unique identification of this object across the cluster of JVMs. After the creation of these fields, the methods of these classes are instrumented. The instrumentation is similar to both instrumentation techniques described above (i.e. for methods of a service object and for other methods). The similarity to the instrumentation of service object’s methods is that in each method a new transaction is created. This transaction is created only if the ReferencedObject field of this object has been initialized. As mentioned above, the initialization of the ReferencedObject implies that the object has left the service’s boundaries, so all updates should be captured. The locked fields of the transaction are referenced with an identifier.
that begins with the reference identifier of the object. This is done because the object has left the service’s boundaries, so it is not accessed anymore with the help of a hierarchy starting from the root object, but with the help of the unique identifier assigned to it when it was first serialized. In this way, all the replicas can locate this object on their local heap and apply the updates done on any of the fields that can be reached by reference from this object. The similarity with the instrumentation of the methods of the non-root objects is that a call context is created for these methods. This call context has always the reference identifier for this object in the index 0 of the call context array and null in the rest indices. This means that the updates of this object’s fields should be captured only if the reference identifier has been initialized, i.e. the object has left the service’s boundaries. The call context in the other indices is always null because the arguments cannot amount to state as they are coming outside the service. So, similarly to the non-root objects’ method instrumentation each time there is a possibly stateful update, which can be a FieldUpdate, a MethodInvocation or an ExternalMethodInvocation the corresponding call context is checked. If the object amounts to state then the update will be added to the currently executing transaction. When we are dealing with synchronized blocks a check for the corresponding call context is done again. In this way, the object is checked if it has left the service’s boundaries and should behave like a replicated object outside the service. If the context is null, the local JVM’s MONITOR instructions are executed, otherwise the cluster-wide lock is acquired. The identifier for the cluster-wide monitor for such objects (and for every field expressed in the hierarchy with respect to this object) begins with the reference identifier for this object and not with this as the identifiers of the monitors that are expressed with an identifier that follows a hierarchy with respect to the service object. This identifier is unique, so no conflict can exist. The same happens with the invocations to wait and notify too.

Initialization of identifiers

Once an object that is part of the service’s state leaves the service’s boundaries, its replicated behavior should be initiated. This is done by initializing the ReplicaNode, Scheduler and ReferencedObject fields of this object. Once these fields have been initialized, the object will be able to capture all the updates happening in its original methods, because the original methods will be the ones called by the callers outside the service’s boundaries. As mentioned above, during the symbolic code analysis of a method of a root object, its return value is checked in case it is something that amounts to state. If this happens, bytecode instructions are injected which initialize the clustering behavior of this object. This is done in every branch of the control flow that returns something that is part of the state. The class of the object will have already been instrumented to contain the appropriate fields that initialize the clustering behavior and implement the appropriate interface. So, the only thing that needs to be done from the injected bytecode here, is to call the appropriate methods of this interface that initialize the three above mentioned fields. The ReplicaNode and Scheduler are initialized to contain a reference to the service object’s ReplicaNode and Scheduler and the ReferencedObject field is initialized to contain a newly created ReferencedObject with the object’s unique reference identifier. The unique reference identifier is fetched from the
ReplicaNode which contains a record of all the objects serialized over the network. If this object has not been serialized yet, it will create a new reference identifier for it now.
Chapter 4

Replication of services

4.1 Related Work

4.1.1 Terracotta

Terracotta [25] is clustering software for Java applications. Terracotta allows threads in a cluster of JVMs to interact with each other across JVM boundaries using the same built-in JVM facilities extended to have a cluster-wide meaning. These clustering capabilities are injected into the bytecode of the application classes at runtime. This is done in a transparent way and there is no need to code against a special clustering API. Terracotta was one of the major inspirations when designing the Virtual OSGi Framework replication mechanisms, therefore some space is devoted to Terracotta implementation.

Terracotta architecture

Terracotta uses a client/server architecture where the JVMs running the clustered application connect to a central Terracotta server at startup. The Terracotta server stores object data and coordinates thread concurrency between JVMs. The Terracotta DSO (Distributed Shared Objects) libraries that escort the original Java classes handle the bytecode instrumentation at class load time and transfer object data, lock-unlock requests at synchronization boundaries and wait, notify requests between the application JVM and the Terracotta server at runtime. However, having one server for all the clustered applications turns the server into a single point of failure and bottleneck. The Terracotta server stores all the object data, which can hurt the scalability of the system, when the number of nodes and clustered applications are increasing.

Injecting clustering behavior and bytecode instrumentation

Clustering behavior is added in the applications by instrumenting the bytecode of the application’s classes during load time. The bytecode of a class is intercepted at load time and examined by the Terracotta libraries. It is then modified according to the configuration provided at Terracotta startup. The mechanism used by Terracotta to track state changes is based on the overloading of PUTFIELD and GETFIELD instructions. PUTFIELD instructions are tracked
to record changes to clustered objects fields. **GETFIELD** bytecode instructions are tracked in order to fetch the objects referenced by the field in question if not already fetched and instantiated on the heap.

To manage thread coordination between multiple JVMs **MONITORENTER** and **MONITOREXIT** instructions are overloaded too, as well as the **INVOKEVIRTUAL** instructions for the various **Object.wait()** and **Object.notify()** methods. **MONITORENTER** signifies the request of a thread for an object’s monitor. A thread will block at this instruction until it is granted the lock on that object. Once the lock has been granted, that thread holds an exclusive lock for that object until the corresponding **MONITOREXIT** instruction is executed. If the object in question happens to be a clustered object, Terracotta ensures that except for blocking to acquire the local lock, it blocks to acquire the cluster-wide lock too. When the thread releases the local lock, it releases the cluster-wide lock too. Cluster-wide locking can be added to methods that were previously **synchronized** locally, or explicitly add clustering to a method that was not previously **synchronized**.

**wait()** and **notify()** methods are also modified, to operate on threads all over the cluster and not on threads of just one JVM.

**Root objects and fine-grained replication**

Clustered objects start at the root of a shared object graph. When a root is first instantiated, the root object and all the objects that can be reached by reference from this object become clustered objects. Their field data are transferred and stored on the Terracotta server. Terracotta uses the notion of transaction which is anchored by the start and end of **synchronized** blocks. Terracotta guarantees that all changes made in all transactions associated with a particular object’s lock in all clustered JVMs will be applied locally before a thread is allowed to proceed past the **MONITORENTER** instruction. The transactions that contain changes to objects contain only the data of the fields that have been changed. These updates are sent to the server and all the JVMs that have objects of the transaction instantiated on their local heap. Because object changes are tracked and propagated on field level, Terracotta does not use Java serialization to propagate the object changes. This approach is much more efficient than replication by serialization because only the values of the fields that have been changed are transferred instead of the entire object graph that Java serialization would propagate. This technique has also an implication on object identities. If Java serialization was used, a changed object would be deserialized in the target location and would somehow have to replace the existing object instance. But with Terracotta technique, a clustered object lives on the heap just like other objects. When changes are made locally to that object, they are made directly on the object on the heap. When a remote transaction arrives, it is applied directly to the existing object that is already on the heap.

**Terracotta use cases**

There are four main use-cases where Terracotta is most effective.

1. **HTTP Session Replication**
   Terracotta Session replication provides high-availability by keeping sessions available across application servers in the event of an application
4.2 Virtual OSGi Framework replication implementation

Some information about the general architecture of the Virtual OSGi Framework architecture would be useful. In the Virtual OSGi Framework, the service registry is distributed. The design of such a service registry is based on an overlay network and more specifically on a DHT, with an architecture very similar to Chord [1]. The services are distributed in the DHT using hashing. There is one coordinator for each service on the DHT and it is selected using the hashing properties of the DHT. Thereby, each node holding a replica will retain a connection with the corresponding coordinator node on the DHT as illustrated in the figure below. The coordinator of a service does the required bookkeeping about each replicated service. More about replicated services coordinators will follow in the next sections.

Figure 4.1: Virtual OSGi Framework Service Registry
4.2.1 System model

The big picture of the system is illustrated in figure 4.2.

Every node in the Virtual OSGi Framework is equipped with the following components.

- **Replica Node**
  The Replica Node adds clustering behavior to a service. One Replica Node is created for each service that is replicated on a certain node. The Replica Node contains all the logic needed to implement the distributed locking and distributed synchronization mechanisms for the service replicas, to send the updates to the coordinator and handle the serialization mechanisms.

- **Communication Manager**
  Each node in the Virtual OSGi framework has a Communication Manager. The Communication Manager is responsible for opening new connections to other nodes and receive requests for connections from other nodes.
Connections are opened to replica coordinators for each replicated service on the node and to other nodes for migrating a service or creating a new replica. The connections are based on the TCPChannel implementation which provides blocking and non-blocking communication behavior.

- **State Manager**
  The State Manager has one instance on each node of the Virtual OSGi Framework. The State Manager offers the `migrate` and `replicate` primitives which migrate and replicate services to other nodes. It holds information about the state of each service and all the information about the inner threads of each service. It is responsible to initiate state capturing of inner threads when a replication or migration request is received and to reestablish or restart the threads when a service is migrated or replicated at this node. State Manager is also responsible for the scheduling of the inner threads of the services. More about the State Manager can be found in the chapter about migration.

- **Scheduler**
  The Scheduler implements the Active Object pattern for the services of a node. One Scheduler per service is created in each node. The Scheduler holds a queue where updates can be queued and is responsible for executing these updates while ensuring the consistency of the service replicas at the same time.

- **Service Record**
  One service record is created for each service at each node. Service records are kept at the nodes where the replicas reside and at the coordinator nodes for these service replicas. The service records are responsible for the bookkeeping required for each service. They are responsible for assigning unique ids at the updates, detecting temporary inconsistencies at service replicas and initiating conflict resolution for them. The records are also responsible for assigning unique identifiers for each object that amounts to state and is allocated on the local heap. They are also responsible for locking and unlocking fields during a transaction execution. Finally, they are managing the cluster-wide locks when it is offered to the node to be managed locally. This mechanism will be described in section 4.2.4.

- **Updates Listener**
  The Updates Listener is the receiver thread of each TCPChannel created on a node. It listens for messages on the endpoint of the socket and performs the required operations according to the parsed message’s needs. It also listens for updates and is responsible for enqueueing the updates in the Scheduler’s Activation Queue.

Similarly to Terracotta there is some kind of ‘server’ responsible for propagating the updates to the nodes holding a certain service replica and coordinating thread concurrency between JVMs. However, there is a significant difference on how this ‘server’ operates. It is not a centralized point like the Terracotta server. Each service has its own coordinator, but there is not one coordinator for all services. The location of a service’s coordinator depends on the hashing properties of the DHT architecture of the Virtual OSGi Framework. Hence, it
42 Replication of services

can be said that the system is fully distributed and peer-to-peer and there is no single centralized point of failure like the Terracotta server. This improves the scalability of the system, since each service can use its own coordinator, when enough nodes participate in the system. Another crucial difference with the Terracotta architecture, is that unlike Terracotta server, the coordinator stores no object data. This reduces the load at the coordinator which serves only as an updates forwarder, a sequencer and a distributed semaphore. Since the concept is significantly different, the name coordinator is used instead of server. In Terracotta, each field of each shared object is stored discretely. Each application-level JVM that participates in the cluster is called an L1 client. This is because, if the application cluster is viewed as a large SMP motherboard, the JVMs are CPU and expensive I/O needs to be avoided to keep our logical CPU from going idle. I/O is avoided by leveraging L1 caching, both in the motherboard as well as in the Terracotta analogy. The Terracotta server can be thought of as a shared L2 for all machines to leverage. Access to L2 costs more than access to L1. This is due to the fact that L2 is across a network connection from all JVMs but the L2 is cheaper than other I/O such as database or messaging that would otherwise be invoked. It is clear that the client/server architecture is a bottleneck and eventually fails to scale, but the presence of a server eliminates network partitioning and restarting problems. The Virtual OSGi framework is using no 'network-attached memory' concept. Instead of using a server, which can be regarded as the primary replica, it uses a peer-to-peer approach and one coordinator for each replicas group. The replicas of a service are not regarded as secondary replicas, or 'cache' which should contact the primary replica to remain up-to-date, but the system is a 'write-any' system. All updates done, are applied only locally and not on the coordinator. Therefore, each replica is able to update a shared object directly and apply the updates done on the local heap before sending them to the coordinator. This has the immediate effect that concurrent updates can be done on two different service replicas and then be sent concurrently to the coordinator which will propagate them to all the replicas.

There is a potential problem with inconsistencies as the two replicas updating the same shared field concurrently cannot have applied the updates in the same order. As mentioned before each replica applies the updates it issues locally and then sends them to the coordinator to be forwarded. It is clear that in the case of two replicas updating the same field concurrently, both replicas will apply first their own update and then the one received from the coordinator, leading to two different states on the two replicas. However, these cases are regarded as infrequent and a conflict resolution mechanism is applied when such events happening. This approach of conflicts resolution was preferred over a possible conflict prevention technique. In the conflict prevention, a mechanism to roll back to a previous consistent state would have to be provided when a conflict appeared. When imagining the variety of operations that can be executed, the only rollback techniques that could be used as conflict prevention would be:

- **Transaction buffer**
  
  In this solution, all the operations during a transaction would be applied in a transaction buffer and not on the real fields. When the 'virtual' execution was over, the node would contact the coordinator and see if any other node had at the same time updated the same fields. If no, the
transaction would be committed directly (and applied to the real fields). If some other node had updated the same fields at the same time, then either the transaction would be aborted or the other update would be applied to the real fields before the local transaction. This technique would lead to great performance overhead, since each transaction would have to be executed twice, once on the virtual fields of the transaction buffer and once on the real fields. Apart from that, this mechanism seems like acquiring a global lock on the fields while updating, like the locking on database transactions.

- **Cache**
  A second solution, could be similar to the Terracotta strategy. In this strategy, the local fields of each replica would be regarded as a cache and the real fields would be located on the coordinator (which would act like the Terracotta server). Like caching, when a field was changed on a replica, it would be tagged as 'dirty' and the updates would be propagated to the coordinator. The coordinator would keep the real fields and apply the updates from the nodes (caches) in a specific order. When a replica wanted to get a fresh and updated copy of the field it should contact the coordinator. This technique however would introduce high communication overhead and long delays while communicating with the coordinator.

- **Blocking technique**
  A third solution would be to have each node block in the beginning of each transaction and communicate with the coordinator. If concurrent requests for begin of a transaction on same fields arrived at the coordinator, then it would serialize the requests and give the acknowledgement for beginning the transaction to the nodes in a specific order, one after the other. This is also a locking technique and would degrade the performance of the services. The begin of a transaction would require waiting at least RTT to get the confirmation from the coordinator.

As mentioned above, the concurrent updates are expected to be infrequent events and as a result locking is not required. The solution of the server was not acceptable either, as we are dealing with a distributed system and we wanted no centralized solution.

### 4.2.2 Service Replicas Coordination

Group or multicast communication [10] requires coordination and agreement. The aim for each of a group of nodes is to receive copies of the messages sent to the group, often with delivery guarantees. The guarantees include agreement on the set of the messages that every node in the group should receive and on the delivery ordering among the group members. In our context, the main concern for all the replicas of a distinct service (communicating through a single coordinator) is to decide on the updates messages to be delivered to all replicas and in the same order. So, unlike basic multicast algorithms delivering messages to group members in an arbitrary order, this lack of an ordering guarantee is not tolerable in our context. Different ordering means different state in the replicas, i.e. inconsistency. The common ordering requirements are total ordering, causal ordering, FIFO ordering and the hybrids total-causal and total-FIFO. A formal
definiton of the three versions (\(g\) being the communication group) of ordering is:

- **FIFO ordering**: If a correct node issues \(\text{multicast}(m, g)\) and \(\text{multicast}(m', g)\) then every correct process delivering \(m'\) will deliver \(m\) before \(m'\).

- **Causal ordering**: If \(\text{multicast}(m, g) \rightarrow \text{multicast}(m', g)\), where \(\rightarrow\) is the happened-before relation induced only by messages sent among the members of \(g\), then any correct process delivering \(m'\) will deliver \(m\) before \(m'\).

- **Total ordering**: If a correct node delivers message \(m\) before it delivers \(m'\), then any other correct node delivering \(m'\) will deliver \(m\) before \(m'\).

Causal ordering implies FIFO ordering, since any two multicasts by the same node are related by happened-before relation. FIFO ordering and causal ordering are partial orderings. FIFO ordering is partial ordering, because it applies only to messages sent by one node. Causal ordering is partial ordering, because it applies only to messages related with the happened-before relation. However, messages can be concurrent. Since messages can sent by many nodes and sometimes concurrently, total ordering is needed in order to ensure consistency among the service replicas.

The basic approach to implementing total ordering is to assign totally ordered identifiers to multicast messages so that every node makes the same decision on the order in which it delivers the messages. There are many methods for assigning unique, totally ordered identifiers to messages. Since, a coordinator for each group of service replicas is being used, the coordinator can assign unique totally ordered identifiers to the messages. The coordinator is called in this case a sequencer. A formal description of this method’s algorithm is provided below (let \(\text{TO-multicast}\) be the primitive of totally ordered multicasting a message). Note that this approach is a slight modification of the sequencer algorithm.

A node wishing to \(\text{TO-multicast}\) a message \(m\) to group \(g\) attaches a unique identifier \(\text{id}(m)\) to it. The messages for \(g\) are sent to the sequencer for \(g\), \(\text{sequencer}(g)\). The node \(\text{sequencer}(g)\) maintains a group-specific sequence number \(s_g\), which it uses to assign increasing and consecutive sequence numbers to the messages that it delivers. Every node delivers the messages it receives with respect to their sequence numbers assigned by the sequencer. If a message with higher sequence number than the one expected arrives, it remains in a hold-back queue until its sequence number is the one expected. If it does not arrive the node asks it from the sequencer in a pull-based approach, as described in section 4.2.6. Every message is acknowledged (reliable multicast).

### 4.2.3 Active object pattern

One of the main motivating problems of this project was high efficiency and performance. However, a major bottleneck of a replicated system is the applying of the updates. The updates are received on the same channel with all the other messages and as a result, the updates messages should be as small as possible and should not delay the messages that come after them. In other words, the updates applying should be decoupled from the receipt of the updates. In the service model of OSGi the applying of the updates should also be decoupled from the invocation of methods on the service objects. The main reasons are:
Updates applied asynchronously
The updates are received by a thread listening on a TCP channel. There are many types of messages received on this channel, some of which are time critical. Applying the updates on the same thread that receives them would block the thread and degrade the message processing efficiency. The updates should be executed on their own thread of execution, while the receiver thread continues parsing messages arriving on the TCP channel. Moreover, applying the updates in their own thread of control facilitates scheduling the updates in an order which is defined by the application’s needs dynamically. It is not needed that the updates are applied in the order in which they are received.

Synchronized access to shared objects becomes simpler
The service objects are accessed concurrently by the threads invoking service methods and the thread that applies the updates. Since there is concurrent access on the same resource, synchronization is needed. However, mutual exclusion using a lock on the object is not the best technique. Moreover, the object updated is clustered, which would mean locking all the replicas of this object. Such coarse-grained mutual exclusion is not needed. What is needed is a schedule according to which the updates will be executed. A field access on this object does not need to block just because some updates should be applied. This access should be blocked only if there are pending updates for this certain field. Updates on other fields could be applied concurrently with the field access. Updates should be applied asynchronously in a separate thread of execution and the client threads accessing the service methods should be blocked only if updates for the fields accessed have been received and not yet applied.

The decoupling can be achieved with multi-threading techniques. The updates should be applied at the services in their own thread of execution. A well known technique for highly efficient concurrent access is the Active Object pattern. This programming pattern decouples method execution from method invocation in order to simplify synchronized access to an object that resides in its own thread of control. The Active Object pattern allows one or more independent threads of execution to interleave their access to data modeled as a single object. In asynchronous processing, the function requesting the value does not block in the body of the function but immediately returns and does the computation or communication out of band. The server informs the client when the service is complete. The reason the Active Object pattern is applied is to improve the QoS of concurrent objects, by allowing an application to handle multiple requests in parallel. The multiple concurrent requests in this context is the request by a client to access an object’s field and the updates received from other replicas that need to be executed. The main components of the pattern are the Activation Queue and the Scheduler. The Activation Queue stores requests for updates execution. The Scheduler runs continuously in its own thread of control, dequeueing requests from the Activation Queue and dispatching them on an object.

In this context, the receiver thread of a communication channel does not block when it issues a request for updates execution, but issues requests to the active object for updates execution. The Active Object in our context is applied as shown in figure 4.3.
As it is shown in figure 4.3, each node in the system is equipped with an UpdatesListener which is the receiver thread of the channel that is connected to the coordinator. This thread is listening for updates messages and other messages that can arrive from the coordinator. When an updates message arrives at the channel, it would be very costly to apply these updates directly at the service's objects. For this reason, a layer of indirection is introduced between the service’s objects and the updates receiver. This layer of indirection is a message queue, where the updates are enqueued as they arrive. The enqueuing of the updates is a very fast operation and the receiver thread can return to a listening state in order to parse the next message arriving on the channel. The message queue is accessed by a separate thread of control. This thread is decoupled from the receiver thread and from the client thread. One such thread and one such message queue exists for each service installed on the system. This thread is called Scheduler, as it schedules the execution of the updates. The Scheduler can serve the updates in a FIFO order as they are enqueued on the message queue, or logic can also be applied on the Scheduler to increase the efficiency of the services. One example of Scheduler's logic could be the following:

- Each thread that accesses the service declares the fields that will be modified during this transaction, when calling BOT for this transaction. The thread could also declare not only the fields that will be accessed in a WRITE mode, but all the fields that are accessed during this transaction together with the mode of access, READ, WRITE, READ/WRITE. All of the fields that are accessed in a READ mode during the transaction should be up to date to ensure consistency, i.e. all of the updates already received for these fields should have been applied. So, each thread initiating a transaction should notify the Scheduler about the fields that it will access in

![Figure 4.3: The Active Object pattern](image-url)
4.2 Virtual OSGi Framework replication implementation

READ mode. The Scheduler then should dequeue all the updates received for the accessed fields, in order to ensure consistency.

- The Scheduler could also schedule the applying of the updates according to the executing transactions, or even drop some updates, by seeing that an upcoming transaction will overwrite some updates that are to be applied. It could also serialize the transactions to be executed in order to avoid conflicts between them.

In the simple model of execution, the Scheduler serves the updates in a FIFO order. While the updates queue is not empty, the Scheduler thread applies updates as they are enqueued by the channel listener thread. However, if the Scheduler does not give priorities to updates, the system can be driven to inconsistent state. If the updates are applied in a plain FIFO order, then a read access can read an obsolete value of the field, because the load on the Scheduler's queue is high and updates that are on the queue for this field have not been applied yet. For this reason, a form of priority should be given to the updates, according to the read requests issued for its fields by the service's clients. The field priorities are achieved by a notification of the Scheduler for the GETFIELD instructions executed by the clients, as shown on the Active Object pattern figure before. This notification (getField) of the scheduler should be very efficient, as it will be executed often. When the Scheduler is notified that a field is accessed, it checks if there are any pending updates on its queue. If no update is pending, the getField returns immediately and the client thread continues its execution. If there is a pending update, this update takes the highest priority, and it is applied immediately. In this case, getField returns once the field is in a consistent state. However, it is expected that in most of the cases the updates for a field will have already been applied asynchronously and getField will return immediately.

Since the updates are applied in a lazy way, the ordering in which they are applied can be different from the one that they were issued. The updates can get higher priority, according to the field accesses. The priority given to some updates requires special handling. This can in some cases lead to an inconsistent state, e.g. when an update depends on a previous update. In such a case, we might have a method invocation that is given high priority because the field on which the method is invoked is accessed by a client. However, it might be the case that one of the objects referenced by the arguments of this invocation amounts to state and this object has some pending updates in the queue. If we do not give the same priority to this field’s updates, the method invocation could have effects on this replica that differ from the effects on the replica that sent this update. This is clearly an inconsistency. Cases like this should be handled by the Scheduler as consistency is paramount. Hence, the Scheduler builds a dependency graph with the updates that are being received. If a getField request arrives for a field, the Scheduler must ensure that the updates should be serialized in such an order, so as to leave all the replicas in the same state. Therefore the Scheduler applies all the dependencies in an update’s dependency graph and then applies the update. In such a way the order in which the updates are applied could differ from replica to replica only for updates which do not depend on each other, i.e. they will have the same effect on the service's state irregardless of the order in which they will be executed.
The Scheduler has to find all the dependencies between the updates and apply quickly all the possible pending updates in cases of field accesses. Therefore, precise and fast identification of the updates, as well as fast execution are crucial points in the design. For this reason, the updates are enqueued in the Scheduler’s queue, with their unique identifier as a key. The unique identifier is also the search key for the Scheduler in order to locate any pending updates. The getField operations are also issued with this identifier. As mentioned in section 3.4.2, the updates are implementing the Command design pattern. In this pattern, a command object encapsulates all the logic needed for the execution of an action. Thereby, the only thing that the Scheduler does is to invoke the execute primitive on the update that is designated being executed next. In this way, the updates propagation is much easier.

4.2.4 Distributed synchronization

Some of the Java virtual machine’s runtime data areas are shared by all threads. This is the data stored in the heap and method area, i.e. instance and class fields. In the context of replicated services, such data is shared among threads of multiple JVMs. Therefore, a distributed locking mechanism is needed to synchronize the access to the shared resources. Locking and thread synchronization is one of the most difficult and tricky concepts in the JVM, so it cannot be easy when the locking and synchronization boundaries exceed the boundaries of a single address space. All the locking semantics of a Java lock should be adapted to hold for the whole cluster of JVMs. The lock is used in a JVM in order to coordinate multi-threaded access to two kinds of data:

- Instance variables, which are stored on the heap.
- Class variables, which are stored in the method area.

This is happening, because the heap and method area are shared by all threads, contrary to the local variables. Local variable reside on Java stack, which is private to a thread. To implement the mutual exclusion capability of monitors, the Java virtual machine associates a lock (sometimes called a mutex) with each object and class. A lock can be owned by only one thread at a time. Concurrent threads cannot acquire the lock, until the owning thread releases it. However, Java locks are reentrant. For each object, the Java virtual machine holds a counter, which memorizes how many times the owner thread has acquired that lock. A lock that is not owned by any thread has a zero counter. When a thread acquires a lock, the counter associated with this lock is incremented to one. Each time the owner thread acquires the same lock, the counter is incremented and each time it releases it, the counter is decremented. When the counter reaches zero, the lock is released and is accessible by other threads.

A lock is requested by a Java thread when it arrives at the beginning of a monitor region. It is released, similarly, when the thread leaves the protected block - no matter how it leaves the block. Once the thread acquires the lock it enters the protected code (critical section). No other thread is allowed to enter a critical section protected from the same lock until the owning thread releases this lock. In other words, the critical section is a piece of code that one thread must be able to execute from beginning to end without another thread concurrently executing a critical section protected from the same lock. When a
4.2 Virtual OSGi Framework replication implementation

A thread tries to acquire a lock that is already owned by another thread, it enters the entry set for this lock. Once the owner thread releases the lock, one thread is chosen from the entry set and the lock is granted to it. The way the entry set is implemented (FIFO, LIFO, priorities) depends on the virtual machine implementation.

All the above semantics should be extended to have cluster-wide meaning. First of all, a lock should be owned by at most one thread across all clustered JVMs. When the owner thread reacquires the same lock, an associated counter should be incremented and when the owner thread releases the lock it should be decremented. Any thread trying to acquire the lock owned by another thread should be added to the entry set for this lock. The entry set should be one for all the clustered JVMs and any thread trying to acquire an already owned lock should be put in this same entry set irregardless of the JVM on which this thread is running. When the counter of the lock reaches zero, the lock should be released and granted to one of the threads in the entry set. This thread could be running in the same JVM with the thread that last owned the lock or in any other clustered JVM. Critical sections should have cluster-wide meaning too, i.e. when one thread is executing code protected by a cluster-wide lock, no other thread in any of the clustered JVMs should be able to enter a critical section protected by the same lock. The entry set implementation depends on the underlying JVM implementation of entry sets. All these cluster-wide lock semantics are implemented by the ReplicaNode construct located at each Virtual OSGi Framework node and the coordinator. The ReplicaNode construct offers the distMonitorEnter and distMonitorExit primitives to the instrumented classes, which manage the cluster-wide locking. During the implementation of the system, two distributed locking policies were used, which are described below.

Altruistic locking

This policy is resembling the locking policy used in the initial Terracotta implementation. It does not expose optimal performance because it requires communication with the coordinator for each enter into a synchronized block and exit from it. As described in chapter 3, each MONITORENTER instruction in the application code is replaced by a call to the ReplicaNode’s distMonitorEnter primitive. The replacement is done when the lock’s owner amounts to state. The implementation of the distMonitorEnter first checks if the thread trying to acquire the lock is the lock owner itself. If this happens, then no other thread in any other clustered JVM can own this lock. As Java locks are reentrant, a counter associated with this lock is incremented without any communication with the coordinator. If the thread trying to acquire the lock is not already owning the lock, a synchronized message is sent to the coordinator. The thread that tries to acquire the lock is blocked until it is granted the lock for this object. The problem here is that the thread will have to block for time at least equal to the round-trip-time of the message between the node and the coordinator, even if this lock is not owned by any other thread at one of the clustered JVMs. This blocking can introduce high overhead because even if the same thread on one JVM tries to acquire the lock more than one times, it will have to wait for the round trip time of the message each time. The blocking behavior is implemented inside the TCPChannel that wraps the Socket used for the connection.
Once the coordinator receives the message, it first retrieves the identifier of
the service and fetches the service’s record, in order to access the locking data
structures for the specific service. If no other thread in the cluster of JVMs is
currently owning the lock, it is granted to the node. Otherwise, the thread that
requested the lock blocks until it is released.

![Figure 4.4: Altruistic Locking](image)

As illustrated in figure 4.4, the delay due to the round trip time of the message
appears even if the lock is not owned by any thread. That is the reason why the
following locking policy was adopted.

**Greedy locking**

Greedy locking is highly optimized for cases where the lock is frequently used by
the threads of the same node. In these cases, there is no point in communicating
with the coordinator every time a thread tries to acquire the lock. It is more
reasonable to let the node whose threads are using the cluster-wide lock to
manage the lock locally. However, this case is not optimal when many nodes
are contending for the same cluster-wide lock. The main principles of this policy
are described below.

Like the previous locking technique, when the application code is invoking
`distMonitorEnter` the `ReplicaNode` is checking if the thread requesting
the lock is the owner of this lock. In this case, a counter associated with this
lock is incremented and nothing else is required to be done. Otherwise, the
`acquireDistLock` method of the service’s record is invoked. The first time this
method is invoked for this lock, it will operate like the altruistic by sending a
synchronous message to the coordinator, which blocks until the node has suc-
cessfully acquired the lock. If the lock is available, the coordinator will check if
it has any requests for the same lock from other nodes. If it does not have, it
will lease the lock to the node, i.e. it will notify the node that it can manage the lock’s state locally, without the need of notifying the coordinator for the lock’s state. If it has requests from other nodes it will not lease the lock to the node, but it will switch to altruistic locking, as above. Each lock can be in one of the following state modes:

- **ACQUIRED** This means that this lock has been leased to this node by the coordinator and it is now owned by a thread on this node.

- **RELEASED** This state mode means that this lock has been leased to this node by the coordinator and it is now owned by no thread on this node, so another thread can acquire this lock.

- **NOT LEASED** This state mode means that this lock is not leased to this node by the coordinator (or it was leased but the lease period has ended) and the ReplicaNode has to communicate with the coordinator in order to acquire the lock again.

When the coordinator receives a request for a lock that is already leased to a node, it will notify the lock owner that the lease period has ended. If no thread at that node is inside a region protected by this lock, the lock will be returned to the coordinator immediately. Otherwise, it will be returned once the thread has left the protected region.

The last case not described is when a thread is trying to acquire a lock that is leased to this node by the coordinator. The lock state will be checked once the request is issued. If the state of the lock is RELEASED, the thread will acquire the lock immediately. If it is ACQUIRED, the thread will block until the current owner has left the protected region. Afterwards, the lock will be granted to the blocked thread immediately if the lease period has not yet ended. Otherwise, the node will have to contend with other nodes to lease the lock again.

It is clear from the above analysis, that this policy will perform better than the altruistic one. If there are multiple requests for the same lock from two or more JVMs this policy will perform the same as altruistic locking since no leasing will be done. The only unoptimized case, is when a node requests a lock that is leased to another node. This will require the coordinator to recall the lock from the node managing it locally and then give it to the node that requested it. This will need time equal to two times the round-trip-time of the message, which is even worse than the altruistic locking. But, it is expected that in most cases the lock will be leased to the node and threads from this node will reuse the lock so the 2 RTTs will be paid off by the fact that the next requests will be managed locally.

**Cooperation**

The second type of thread synchronization supported by Java’s monitor is cooperation. Cooperation means that threads are working together towards a common goal. It is supported in the Java virtual machines via the wait and notify methods of class Object. A thread that currently owns the monitor can suspend itself inside the monitor by executing a wait command. When a thread executes a wait, it releases the monitor and enters a wait set. The thread will stay suspended in the wait set until some time after another thread executes a
notify command inside the monitor. When a thread executes a notify, it con-
tinues to own the monitor until it releases the monitor of its own accord, either
either by executing a wait or by completing the critical section. After the notifying
thread has released the monitor, the waiting thread will be resurrected and will
try to reacquire the monitor.

Since the behavior of the monitor is extended to have distributed semantics, the
cooperation operations of the monitor have to be extended too. As mentioned
in chapter 3, the wait, notify and notifyAll method invocations (on a monitor of
an object that amounts to state) in the application’s bytecode are replaced by
the distributed ones. Now it is time to have an inner look in the implementation
of the distributed cooperation operations.

First of all, it should be stated that the cooperation mechanism has to commu-
unicate always with the coordinators and cannot be managed locally. The wait
set is clustered and it can contain threads from all the JVMs appearing in the
cluster of JVMs. As a result, when one or all the threads that are waiting on a
clustered monitor are notified, it cannot be known if the thread(s) to be noti-
fied are running on the local or a remote JVM. Consequently, every notify(All)
operation should notify the coordinator, which holds the clustered wait set.
Likewise, when a thread on a JVM is suspended to wait on a clustered monitor,
it cannot be known if the thread that will notify it, is located on the same JVM
or a remote one. Consequently, every wait operation should communicate with
the coordinator and add the suspended thread in the clustered wait set.

4.2.5 Updates propagation

The updates created by the replicas are sent to the coordinator and the coordi-
nator after serializing them and assigning a unique sequence number to each of
them, propagates them to all the replicas in the same order, using totally ordered
multicast. During the propagation, the coordinator sends the UpdatesMessage
4.2 Virtual OSGi Framework replication implementation

as it is to all nodes except for the node from which this update was originated. This node has already applied this update locally, so the only thing it is needed to be known by it is the sequence number assigned to this update. Consequently, the message sent to the originating node is an UpdatesMessage containing the service identifier, the node identifier of the originator, the old sequence number of this update and the new one, if modified. The updates are probably the most frequent messages exchanged, so they must remain as small as possible. The updates messages are sent using a modified version of Java serialization. The most 'expensive' operation of Java serialization is the writeObject operation, because it can lead into serializing a large object graph without this even being needed. Apart from that when an object is serialized more than one times, a new object is instantiated on the heap of the serialization destination, leading to duplicates. Moreover, most of the updates are just an update on an object’s field. This object can contain many other fields which again contain many fields with references to other objects leading to a very deep object graph. Serializing the whole object would serialize the whole object graph, which would lead in a very large and expensive message. As a result, the following optimization techniques have been applied.

Fine-granular updates

When adding an update to a transaction, only the identifiers and the data of the fields that have changed are communicated through the network. Updates are defined at field level and no other data from the object are sent during updates propagation. This approach is much more efficient because it only moves the data that has changed across the cluster instead of entire serialized object graphs.

Object identity and serialization

Every object that is serialized and propagated over the network to all other replicas is automatically assigned a unique reference identifier across the cluster of JVMs. This is done for two reasons. First of all, this prevents from serializing the object again if it is being used for another time and sending a reference identifier instead. This leads to much smaller messages, as it is frequent to send the same value across the network again. The second reason for which these reference identifiers are created is to retain the object identity. If the same reference is used in an update for a second time and the object was serialized again, this would lead to an instantiation of a new object in the destination heap. So, there would be two objects on the heap, instead of one object and two references to it. Furthermore, equals operations would no longer operate correctly. Object identity is also handled to some extent by Java serialization. However, solely relying on the caching of the Java serialization protocol would break whenever the object stream is reset. Additionally, other events could break the caching as well. For instance, when node 1 sends an update to node 2 that enqueues an object into an ArrayList. If node 2 then removes this object from the ArrayList and wants to propagate this update to node 1, it should somehow refer to the object that is instantiated on the heap of node 1. Without the use of reference identifiers, this case could not be handled even with the use of stream caching in Java serialization, because the cached reference would be
sent on the object stream from node 1 to node 2 and not in the other way round (since this object was not serialized in the object stream from node 2 to node 1).

Finally, it is known that an object can be serialized with the help of Java serialization, only if it implements the `Serializable` interface. However, most of the objects can be serialized, i.e. they do not make use of resources specific to a single JVM, but they cannot be serialized with the use of Java serialization because they do not implement this certain interface. Hence, a deeper serialization mechanism was needed. This was implemented with the use of JVMTI (see section 7.1), which has access to all the information of an object. The serialization mechanism checks if the object to be sent through the network already implements the `Serializable` interface. If it does, the methods of Java serialization are called, since no special treatment is needed. If not, the native serialization function reads all the instance fields of the object and serializes them. At the destination, the native deserialization function decides if a method of Java serialization has to be called or if it should read all the serialized values of the fields and instantiate a new object.

### 4.2.6 Conflict resolution

As already mentioned in a previous section, the updates occurring on a node are first applied locally and then propagated to all other replicas by sending them to the coordinator. This policy strengthens efficiency, but it can lead to inconsistencies when two or more nodes are updating the same field concurrently. The transactions containing concurrent updates are serialized (i.e. ordered in a serial schedule) at the coordinator with the use of unique sequence numbers and totally ordered multicast. This does not solve the problem in its entirety though, as every node concurrently updating the same field has already applied its update to its local fields before sending the update to the coordinator. If the coordinator decides that the update of a node should be applied after all the other updates, the node that has applied this update locally will be in an inconsistent state. It will have applied this update before applying the preceding updates in the serialized transactions list, under the wrong assumption that no other node was updating the same field concurrently. This is not a problem for assignments on fields, since the last update will overwrite the previous ones. Applying all the field assignments in the serialized order, or applying only the assignment with the greatest sequence number has the same result. So, the only thing that a node should do is to keep track of the sequence identifiers of the field assignments and not apply any concurrent field assignments until its own update has been assigned a sequence number. All the field assignments with sequence number smaller than the one assigned to the update that has been applied locally would have been overwritten anyway (cancellation).

Unfortunately, the problem is not solved with the cancellation technique in cases of updates that are method invocations. This type of updates occurs when a method is invoked on an instance of a JDK class and this instance is referenced by a field of the service. The code of the JDK class cannot be instrumented, since it will already have been loaded by the bootstrap classloader, when the service classes are instrumented. Therefore, these updates are propagated by invoking the same method on all replicas. One method invocation on the same object does not overwrite the result of another, but they are altering the state of the object.
and they are leading to different state when they are applied in different order. For this reason, the system is equipped with a conflict resolution mechanism. First of all, it should be stated that when a transaction is executed on a node and updates arrive at this node for one of the fields that the transaction has locked, these updates are queued. Once the transaction has been terminated, the updates queued are parsed and the conflicts, if any, are analyzed. In cases of conflicting field assignments the cancellation technique is applied. There are two cases where conflicts can arise:

1. The node performed a method invocation on a field while executing a transaction and another method invocation update for this field arrived during the transaction execution. It is sure that the method invocation originated from this node will be assigned a greater sequence number than the one received. However, the method invocation originated from this node has already been applied without applying first the method invocation received.

2. The node has sent a method invocation update to the coordinator and it receives a method invocation for the same field without receiving first a sequence number for its own update. These are truly concurrent updates and the method invocation sent from this node is going to have its sequence number modified, as it is serialized to come after the one received. In this case the node has again applied its method invocation before applying the one preceding it in the serialized transactions list. It is clear that there is a conflict again.

Once the node detects a field conflict, it notifies the local service record for this conflict. The corresponding field is marked as conflicting and any updates received for this field from now on are queued to be applied once the conflict is resolved. Afterwards, the node notifies the coordinator about the conflict. All updates for fields from now on are not acknowledged by this node. The coordinator initiates a conflict resolution process. It waits until one of the replicas has acknowledged the last sequence number generated for this field. The node that acknowledges the last sequence number may be a node that did not originate a concurrent update for this field and applied the updates in the correct order, as they were received by the coordinator (totally ordered). If such a node does not exist, the node that acknowledges the last sequence number would be a node that originated one of the concurrent updates, but this update was assigned the smallest sequence number among the concurrent updates. In other words, the coordinator decided that the update of this node should be applied first, so this node that applied this update locally first, because it originated it, is the 'winner'. Either way, there will always be a node that has a conflict-free replica of the field. This node will send the conflict-free value to the coordinator which will propagate it to all nodes. Once a node receives this value, it will enqueue it in the Scheduler of this service, and it will check if any updates for this field were received and queued while the conflict was being resolved. If any updates with sequence number greater than the version of the conflict-free value have been received, they will be enqueued at the scheduler, too. The field is marked as conflict-free again.
Chapter 5

Migration of Services

Migration is a mechanism to continue the current execution of a service in another location in a distributed system. Java platform allows the creation of objects in memory. However, all of these objects exist only as long as the Java Virtual Machine remains running and they are reusable only on the Java Virtual Machine that they have been created. This problem is solved by the Java Object Serialization mechanism. With object serialization, objects can be flattened and used on different JVMs. One could argue that with the use of the Java Serialization API a service can be migrated to another machine, retain its state and remain accessible and operational on the new machine. This is only partially true. In Java, the state of an object involves the following:

- Program state: This is the byte code of the object’s class. The byte code of the object’s class can be easily transferred. It can be uploaded to the new machine and loaded through user-defined class loading.

- Data state: This is the content of the instance fields of the object. This is handled by the Java Serialization mechanism as long as the object referenced by the field implements the Serializable interface of Java.

- Execution state: A Java object is executed by one or more JVM threads. Each JVM thread has its own program counter and a private stack. JVM threads cannot be transferred. Indeed, it would not make any sense if they were. For example, thread running in one JVM would be using that system’s memory. Persisting it and trying to run it in another JVM would make no sense at all.

Although code migration and data migration are strongly supported in Java, thread migration is not supported by current Java technology [29] and furthermore, the Java language does not provide an entry point to a thread’s internal state. It does not provide any mechanisms for capturing a thread’s state and reestablishing it later. Hence, service migration becomes much more difficult because execution state of the service’s threads is required. As a result, a thread serialization mechanism has to be implemented. The following section puts more light onto the internals of threads and thread state in Java [27].
5.1 Java thread internals

5.1.1 The Program Counter

Each thread of a running program has its own Program Counter (PC), which is created when the thread is started. The PC is one word in size and can therefore hold both a native pointer and a returnAddress. The Java Virtual Machine defines an abstract notion of a word that has a platform-specific size. While a thread is executing a Java method, the PC contains the address of the current instruction executed by the thread. An address can be a native pointer or an offset from the beginning of a method’s base address. If a thread is executing a native method, the value of the PC is undefined.

5.1.2 The Java Stack

A new thread started by the JVM creates a new Java stack for the thread. The Java stack stores the execution state of a thread by storing frames. The two primitive operations on the stack are pushing and popping frames. For the current method there have been created a current frame, which is the stack frame on the top of the stack and a current constant pool. As it executes the method, the Java virtual machine keeps track of the current class and current constant pool. When the virtual machine encounters instructions that operate on data stored in the stack frame, it performs those operations on the current frame. Once a thread invokes a Java method, the virtual machine creates a new frame and pushes it onto the thread’s Java stack. This frame has now become the current frame. As the method executes, it uses the frame to store parameters, local variables, intermediate computations and other data. All of the data on a thread’s Java stack are private to that thread. A thread has no way of accessing or altering another thread’s stack. The Java language does not give an entry point for a programmer to the a thread’s stack either.

5.1.3 The Stack Frame

The stack frame consists of three parts: the local variables, the operand stack, and the frame data (see figure 5.1.3). The sizes of the local variables and the operand stack depend on the method and are determined at compile time. The frame size is stored in the class file data. Once the Java virtual machine invokes a Java method, this class-file data is read and a frame of the appropriate size is created and pushed on the stack. This stack frame is destroyed when the method returns.

Local Variables

The local variables in a Java stack frame are organized as a zero-based array. When a method accesses a local variable it accesses it by using its index in the array of locals. Variables of type int, float, reference and returnAddress occupy one entry in the local variables array. Values of type byte, short and char are converted to int before being allocated on the local variables. Values of type long and double occupy two consecutive entries in the array. When referring to a local variable of type long or double, it is accessed by using the index of the first of its two entries in the local variables array. All values in the local variables
are word aligned. Double-entry *longs* and *doubles* can start at any index. The local variables section of a frame contains a method’s parameters and the local variables created by the method. The compiler places the parameters into the local variables array first in the order in which they are declared. In case of instance methods, the first local variable contains a hidden *this* reference. Instance methods use this reference to access the instance object data upon which they were invoked. Static methods are not invoked on objects, so it’s not possible to directly access a class’s instance variables from a class method because there is no instance associated with the method invocation.

**Operand Stack**

The operand stack is organized as an array of words, similar to the local variables. The difference is that the values are not accessed from an array-like structure as for the local variables, but from a stack structure by pushing and popping values. Like local variables, values of type *long* and *double* occupy two slots in the operand stack and values of type *byte*, *short*, and *char* are converted to *int* before pushing them on the operand stack. Other than the program counter, which cannot be directly accessed by instructions, the Java virtual machine has no registers. The Java virtual machine is stack based, rather than register based, because its instructions take their operand from the operand stack rather than from registers. Instructions can also take values by values directly following the opcode of the instruction or from the constant pool. However, the Java virtual machine’s instruction set main focus is the operand stack. The Java virtual machine uses the operand stack as a work space. Many instructions pop values from the operand stack, operate on them and then push the result.

**Frame Data**

In addition to the local variables and operand stack, the Java stack frame includes data to support constant pool resolution, normal method return, and exception dispatching. This data is stored in the *frame data* portion of the Java stack frame. Many instructions in the Java virtual machine’s instruction set
refer to entries in the constant pool. Some instructions push constant values of type \textit{int}, \textit{long}, \textit{float}, \textit{double} or \textit{String} from the constant pool onto the operand stack. Some instructions use constant pool entries to refer to classes or arrays to instantiate, fields to access, or methods to invoke. Other instructions determine whether a particular object is a derivative of a particular class or interface specified by a constant pool entry.

5.2 Related Work

Systems providing Java thread migration can be implemented at different levels. Java thread serialization mechanisms are characterized by four properties:

- The \textit{genericity} of thread serialization, i.e., the ability to adapt it to different uses such as mobility, persistence,
- the \textit{completeness} of the accessed thread state,
- the \textit{portability} of the serialization mechanism across different Java environments,
- and the \textit{efficiency} of the mechanism, i.e., its impact on the performance of thread execution.

The techniques used can be classified in four basic approaches:

- **Modification of the Virtual Machine:** the virtual machine is modified to export execution information. This approach has the advantage that it does not affect the performance of thread execution at all and it results in a complete access of thread state. On the other side, it has the major disadvantage of lack of portability with respect to the standard VM. It is very difficult to maintain complete compatibility with the Sun Java Specification.

- **Instrumentation of the application’s source code:** this technique consists in the use of a preprocessor (a source code compiler). The preprocessor inserts source code responsible for capturing and restoring the execution state. The main problem with this technique is that the source code must be available, which is not always possible, like the case where libraries are used. Another disadvantage is its impact on the performance of the application’s execution and the space overhead caused by the insertion of code.

- **Instrumentation of the application’s bytecode:** in this approach, the code for capturing and restoring the execution state is inserted directly in the application’s bytecode. As well as the source code instrumentation approach, some time and space overhead is generated by the injection of additional code in the original code. However, the overhead is generally lower than the source code approach. Another advantage is that, on the bytecode level, an extended set of instructions can be used, like the \textit{goto} instruction.
5.2 Related Work

- **Modification of the Java Platform Debugger Architecture:** the Java Platform Debugger Architecture (JPDA) is part of the virtual machine specification. Using JPDA, runtime information about application can be accessed in debug mode. This can be used to perform transparent migration. Since the JPDA does not provide all information necessary for transparent migration, some modifications in the JPDA core are necessary. Moreover, this approach does not allow the use of JIT compilation, raising the application execution time, generating the highest overhead of all approaches.

Regarding the existing solutions, the thread serialization systems based on a JVM-level implementation verify the completeness requirement but lack in efficiency and portability. The thread serialization systems proposed at the application level are portable but they are neither efficient nor complete. The most intuitive approach to access the state of a Java thread is to add new functions to the Java environment in order to export the thread state from the JVM. In the Sumatra [2], Merpati [24], ITS [6] and CIA [14] projects the JVM is extended with some mechanisms that export the state of the thread in a serializable form which can be transferred and reestablished at a destination location. But they rely on a specific extension of the JVM and cannot be portable to existing implementations.

To avoid the drawback of non-portability, other projects provide a solution at the application level. In these approaches, the application code is transformed by a pre-processor prior to the execution in order to attach a backup object to the Java program which will contain the thread state in a serializable form and add new instructions in the program to capture its execution state. The backup object can be serialized using Java serialization. Wasp [12] and JavaGo [23] use a source code pre-processor. This solution is incomplete since the source code is not always available. Brakes [26] and JavaGoX [22] are more complete because they use a bytecode pre-processor. The advantage of these approaches is that they are portable across all JVM implementations. However, they are not able to access the entire execution state of a Java thread, because some part of the state is internal to the JVM.

The main conclusion from the study of the above approaches is that none of these techniques can be regarded as complete and optimal. One should consider the system’s needs in order to decide if portability will be sacrificed in favor of full state capturing or vice-versa. Moreover, whatever the level of implementation (JVM or application), all existing solutions impose a performance overheads on threads. The overhead on thread performance can be very high (+335%, +340%) in the above JVM-level solutions, as none of them supports Java JIT compilation. This problem is solved by CTS [7]. In this project, a Java thread serialization mechanism is implemented within Sun Microsystem’s JVM. The lessons learned from this experiment were that it is possible to extend the Java Virtual Machine with thread serialization, mobility and persistence facilities without redesigning the whole VM and that this can be done without any performance overhead. This is possible by the use of a type inference technique which separates thread serialization from the JVM interpreter and by the use of dynamic de-optimization techniques which allows thread serialization to be compliant with Java JIT compilation. However, non-portability remains an issue. The overhead in the application level approaches cannot be avoided ei-
other, as they add more instructions in the application code. A summary of the studied techniques is depicted in table 5.1.

### 5.2.1 Brakes

Brakes [26] is a system developed to capture and reestablish the state of Java threads. It was the main inspiration for the Virtual OSGi Framework migration implementation. The thread serialization mechanism is implemented by instrumenting the original application code at the byte code level, without modifying the Java Virtual Machine. It is mainly used in the context of middleware support for mobile agent technology. It consists of two parts, a) a ByteCode transformer (based on version 1.4 of the BCEL) which instruments Java class files so as to be able to capture their current internal state and b) a small framework which uses the ability of the ‘patched’ classes to allow Java threads to pause and resume whenever desirable. Brakes can only serve internal migration requests, i.e. migration is initiated by the agent itself. Building on Brakes, a system called Distributed Brakes was developed for serializing the execution state of distributed applications, that are programmed by a conventional Object Request Broker. Distributed Brakes is used for repartitioning Java applications at run-time. Distributed Brakes can serve external migration requests, too.

### 5.3 Virtual OSGi Framework Migration Implementation

#### 5.3.1 Overview and design decisions

After an overview of the requirements of a concrete migration mechanism, a full description of the migration mechanism used in the Virtual OSGi Framework can be presented. The Virtual OSGi Framework is a system intended for usage in large-scale distribution scenarios and over heterogeneous platforms and computer networks. The migration will be initiated autonomously by the framework after decisions about performance, load etc. Therefore, the migration mechanism should exhibit the following requirements:

<table>
<thead>
<tr>
<th>System</th>
<th>Implementation approach</th>
<th>Portability</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wasp</td>
<td>Application – level</td>
<td>Yes</td>
<td>No (Overhead)</td>
</tr>
<tr>
<td>JavaGo</td>
<td>Application – level</td>
<td>Yes</td>
<td>No (Overhead)</td>
</tr>
<tr>
<td>Brakes</td>
<td>Application – level</td>
<td>Yes</td>
<td>No (Overhead)</td>
</tr>
<tr>
<td>JavaGoX</td>
<td>Application – level</td>
<td>Yes</td>
<td>No (Overhead)</td>
</tr>
<tr>
<td>Sumatra</td>
<td>JVM – level</td>
<td>No</td>
<td>No (Overhead)</td>
</tr>
<tr>
<td>CIA</td>
<td>JVM – level</td>
<td>No</td>
<td>No (Incompatible with JIT)</td>
</tr>
<tr>
<td>Merpati</td>
<td>JVM – level</td>
<td>No</td>
<td>No (Incompatible with JIT)</td>
</tr>
<tr>
<td>ITS</td>
<td>JVM – level</td>
<td>No</td>
<td>No (Incompatible with JIT)</td>
</tr>
<tr>
<td>CTS</td>
<td>JVM – level</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 5.1: Existing systems evaluation
5.3 Virtual OSGi Framework Migration Implementation

- Transparency
- Portability
- Minimum overhead during normal execution
- Strong migration properties

The transparency requirement is needed because the Virtual OSGi Framework should appear as a local framework to the outside world. This means that the services on the Virtual OSGi Framework should exhibit the same behavior as on a traditional, local OSGi framework. Portability is also paramount, because the Virtual OSGi Framework is expected to treat the whole network as a virtual framework and as a result the migration mechanism should be portable to all the platforms participating in the virtual framework. As a result, custom VM implementations are not applicable since they contradict with the open world assumption.

A migration mechanism is not acceptable if it hurts the services’ runtime performance significantly. The overhead of the migration process itself is secondary. What is most significant is to have the service operate as closely to its normal operation (without thread state capturing) as possible. The techniques adopted to capture the state of a service’s thread should not cause the service perform much worse while remaining at the same location.

Migration can be classified into two categories according to its properties. Weak migration is happening when the programmer has to provide explicit code to read and reestablish the state of an application, strong migration is when this is handled by the framework on which the application is running. From the above requirements, it is obvious that a solution based on the VM modification is unacceptable and an application-level solution had to be chosen. A VM-level solution could help in keeping the overhead at minimum levels and have full access to the thread’s state, but it would hurt portability. Portability was regarded as the most significant property of our system, so some additional overhead had to be tolerated. This performance overhead can be minimized even with application-level solutions by the use of native techniques, like the Java Native Interface (JNI) [19] and the JVM Tool Interface (JVMTI) [18] available on Java 5.0. By choosing an application-level migration approach the following benefits arise:

1. bytecode transformations integrate the required functionality transparently into existing Java applications. A custom class loader can automatically perform the bytecode transformation at load-time.

2. the serialization mechanism does require no modifications of the JVM. This makes the implementation portable on any system, as long as standard JVM is installed on that system.

However, the limitation of this approach is the performance overhead as mentioned above. The latency experienced between a migration request and the end of the migration operation depends on the time spent during method invocations. Therefore, if a method invocation takes long time, for example an RPC-like call, the migration request can take time in order to be completed. This is not directly a shortcoming of this approach, but rather a shortcoming
of the RPC-like programming model. After all, performing blocking calls on remote objects is only feasible on a reliable, high-bandwidth and secure network.

5.3.2 Implementation of thread serialization

The thread serialization mechanism is implemented by extracting the state of a running thread from the application code that is being executed in this thread. To achieve this, a bytecode transformer was implemented that instruments the application code by inserting appropriate code blocks that do the actual state capturing and reestablishment. For this, ASM \cite{asm} was used which is an all-purpose Java bytecode manipulation and analysis framework, that offers a programming interface for bytecode engineering. Running threads can capture their state at safe points, which are all the points just before a method invocation in the application code (method entry points).

Thread execution model

As already mentioned, threads are not serializable. Therefore, their state has to be captured in a serializable form in order to be reestablished at the migration destination. For this reason, a higher level object for each thread of a service is created. These constructs are called Tasks and enclose a reference to the code that the thread is executing, i.e. an implementation of Runnable interface. Tasks are wrapper objects for the code executed by the thread. A Task encapsulates a JVM thread that is used for executing that task. As such, a task’s execution state is the execution state of a JVM thread in which the task is running. The JVM thread is started and scheduled by the StateManager, which is running on every node in the Virtual OSGi Framework. The StateManager is also responsible for holding a record with all the threads that have been scheduled for each service. In this way, the StateManager can put the threads into capturing state when a migration or replication request arrives for a specific service. When the StateManager starts a thread, the task is assigned a JVM thread for executing itself. The different threads can run concurrently and rely on the efficient JVM implementation for context switching support. The Tasks have one-to-one mapping with a service’s thread, i.e. for each thread of the service, a new Task is created.

Each Task is associated with several boolean flags that represent its execution state. A Task can be in five modes of execution:

- **isCapturing**
  This means that the thread is now capturing its internal state. This flag is set once the migrate or replicate primitive is called on the StateManager. Once the thread has captured its execution state in a serializable form, it terminates.

- **isSwitching**
  This flag is set when the capturing phase is initiated in an object that is accessible by reference from the thread. Its full meaning will become more clear in the following sections.

- **isRestoring**
  This flag is set when the thread enters the phase of reestablishing its pre-
viously captured execution state. It is set once a migration or replication operation has been completed and the thread is resuming its execution.

- **isAwaken**
  This flag means that when the migration or replication request was issued, the thread was waiting on a replicated object’s monitor (distributed monitor). In this case the thread is awaken and after migration is complete it should resume its state, i.e. it should continue waiting on the replicated object’s monitor. The `wait()` operations on replicated objects do not depend on Java `wait()` implementation, so they can be manipulated as needed.

- **isRestarting**
  This flag is set when a thread has captured its internal state, due to a call to the `replicate` primitive. In this case the thread should restore the values of its fields on the replication destination, but it should not resume its execution from the point it was stopped while capturing, because in this way the operations would be executed twice.

When none of these flags is set, the thread is executing normally. The **Task** stores also some other valuable information required for state reestablishment.

1. **Class name**
   The class name of the thread object is stored inside the **Task**. The class name is used in order to load the thread’s class in the replication or migration destination. After the class is loaded a new instance is created and stored on the local heap.

2. **Instantiation arguments**
   The arguments used for creating a new instance are stored too. These are used in the replication or migration destination in order to create the same object with the same initialization parameters on the heap of the thread’s new location. The correct constructor is located, the arguments are passed and a clone of the initial thread is created.

**Capturing execution state**

Whenever the **StateManager** receives a service migration or replication request, it should initiate the service’s threads execution state capturing. In order to do this it has to know all the threads that are associated with a certain service. For this reason, each service object registers the **Tasks** it creates. This is done by intercepting the instantiation and call to the `start()` method of Java Threads, or subclasses of `java.lang.Thread`, or any other implementations of the `java.lang.Runnable` interface and injecting the additional bytecode instructions. An example instrumentation follows to illustrate the mechanism. There is a **ServiceImpl** class containing an inner class **Worker** which extends `java.lang.Thread` and its constructor looks as follows.

```java
new Worker().start();
```

In the above code, a new **Task** needs to be created for this thread, registered with the **StateManager** and scheduled for execution. After instrumentation the above code will look like this.
Object threadArgs[] = new Object[1];
ServiceImpl tmpArg0 = this;
threadsArgs[0] = tmpArg0;
Thread tmpThread = tmpArg0.new Worker();
StateManager.scheduleAndRegister(new Task(tmpThread, threadArgs), this);

The creation of the new Worker instance passes it a hidden argument of a reference to the outer class. This is intercepted by the bytecode transformer and is stored in order to be able to reinstantiate the thread at the destination location. After that, a new Task is created passing the thread as an argument and it is registered with the service at the StateManager which will schedule it with a new JVM thread. In this way, the StateManager has a mapping between the services running on this node an the Tasks each of them has created. When it receives a migration or replication request for a certain service, it will initiate the capturing process for all the service’s threads by setting the isCapturing flag of the Tasks of the threads. The capturing process executes the state capturing code block for every frame on the stack. The frame for which the state capturing code block has been executed is then discarded by terminating the execution of the corresponding method through an injected return instruction and at the same time initiating the state capturing process for the previous frame on the stack. For each method on the stack, an artificial program counter is saved, which is an index that refers to the last nested method invocation instruction in this method. The same process is then recursively repeated for each frame on the stack, until the frame of the run() method is reached, where the thread terminates through an injected return instruction. The capturing code blocks inserted are of three types.

• Capturing initiator block

The first type of capturing code blocks is associated with the isCapturing flag of the corresponding Task. Capturing blocks of these types are added just before each INVOKEVIRTUAL, INVOKEINTERFACE or INVOKESPECIAL bytecode instruction. These blocks have a check of the isCapturing flag. This is the place where the capturing procedure begins after the StateManager sets the isCapturing flag. An external migration or replication request can be tracked by a thread only in the capturing initiator blocks. This differentiation between the capturing initiator code blocks and the capturing propagation code blocks is done in order to distinguish between a method that had nothing in its body executed, because the capturing process started just before this invocation and a method that was partially executed and the thread execution was suspended before one of the nested invocations in that method. So, the method that was not executed at all will capture its state in the capturing initiator block and all the others, with their frames on the stack, in the capturing propagation block. This differentiation will be illustrated in the example of the next section. The capturing initiator blocks set also the isSwitching flag of the Task in order to notify the calling method that a capturing process has been initiated. After that it executes a injected return statement, destroys its frame and passes control to the calling method, to capture its frame too.

• Capturing propagation block

This type of capturing blocks is associated with the isSwitching Task
flag. These blocks are injected just after each `INVOKEVIRTUAL`, `INVOKEINTERFACE` or `INVOKESTATIC` bytecode instruction. The capturing propagation block are injected only after method invocation on instances of service’s objects whose bytecode can be instrumented. If for example there was a method invocation on a Java HashMap, then it would have no meaning to add a capturing propagation block after it, as the HashMap’s bytecode will not be instrumented and it cannot have initiated a capturing process.

- **Interrupted capturing block**
  The third type of the capturing code blocks is associated with the `isAwaken` flag of Task. These code blocks are injected just after the `INVOKEVIRTUAL` bytecode instruction of `wait()` method invocations on a cluster-wide monitor. If the thread is waiting on a cluster-wide monitor once a migration or replication request is issued, the `StateManager` will interrupt it and set the `isAwaken` flag of its Task. Then, the thread will fall into the interrupted capturing block, capture its state and propagate the capturing process to the calling method, if any.

After initiating the capturing process, the `StateManager` will wait for all the threads of the service to terminate. When they have terminated, the `StateManager` will know that their state has been captured and it can proceed with the migration or replication.

**Reestablishing execution state**

Once the transferring of the data needed for migration or replication has completed, the `CommunicationManager` of the target node will set the state of all tasks included in the service as restoring by setting the `isRestoring` flag of the corresponding `Tasks` in cases of migration and the `isRestarting` flag in cases of replication. However, the actual reestablishment of the threads’ execution state is initiated when the `StateManager` schedules the task. This means that the `StateManager` will invoke the `resume` primitive on each `Task` transferred together with the service. The `resume` primitive creates a new instance of the thread that is enclosed inside the `Task`. The code of the thread is sent as a byte array together with the service and is loaded as a class in the destination, with a user-defined class loading mechanism. The new thread object will be referenced by the `Task`, which will call `run()` on it, once the `Task` is scheduled. Finally, the `StateManager` will start a new JVM thread for each of the `Tasks` transferred with the service and run the `Task` inside this thread. Subsequently, the `Task` is scheduled and starts the execution of the thread’s instructions. It will first execute the additional code that is injected at class loading time by the bytecode transformer to restore its state.

**Bytecode transformations**

As mentioned above, in order to perform the appropriate bytecode transformations, the Node API of ASM Java bytecode manipulation and analysis framework [8] was used. After the symbolic analysis described above is complete, all the updates taking place inside a thread’s method and all the nested method invocations appearing inside the thread method have been detected. As mentioned above, for every method analyzed a new `MethodAdapter` is created, which
is a custom subclass of ASM’s \texttt{MethodNode}. The \texttt{MethodAdapter} holds two flags that are important during the bytecode transformations for thread serialization.

- \texttt{isThreadInstr} This flag indicates that a method is accessed from a thread’s method directly or in more than one steps. This means that thread state capturing and restoring code blocks have to be injected in this method. This flag also means that the method (and all methods after that in the call hierarchy) contains no updates. So, while injecting bytecode in this method state capturing and restoring code will be injected but no replication instrumentation will be needed.

- \texttt{isThreadAndReplInstr} This flag indicates that the method is called by a thread’s method and that it also contains updates (or some other method after that in the call hierarchy). In this case, things are a little more complicated. The original method has to be instrumented for thread state capturing, but the same method with the additional transaction argument has to be instrumented for thread state capturing too.

In each of these cases, information about the stack state on the safe points (nested method invocations) is needed. So, when there is a method with one of these flags set, the stack frames constructed by the \texttt{SymbolicAnalyzer} during the analysis are stored. In this case, a \texttt{ControlFlowAnalyzer} (which subclasses ASM’s \texttt{Analyzer}) is used instead of the plain analyzer. The \texttt{ControlFlowAnalyzer} checks the instruction list and constructs a control flow graph. By use of this control flow graph, it is known for each nested method invocation instruction inside the method, which fields can be read after the invocation returns, in any possible control flow. This is needed in order to check which fields should be captured and restored on every safe point where the execution may be suspended. This can save some field data transferred while sending the \texttt{Context} object to the migration or replication destination. By use of the frame states, at each safe point the local variables created and the stack condition are known, in order to push all the local variables and stack operands on the stack of the corresponding \texttt{Context} object, together with the, reachable by the control flow, fields. Now, all the information needed for the thread state capturing code injection is available. The thread state capturing bytecode instrumentation is done by the \texttt{MigrationInstrumentor}. The \texttt{MigrationInstrumentor} is a field of the \texttt{BytecodeInstrumentor} described above. It is instantiated only when a \texttt{MethodAdapter} with \texttt{isThreadInstr} or \texttt{isThreadAndReplInstr} flag set is instrumented. The main operations of \texttt{MigrationInstrumentor} are:

- \texttt{addCapturingCodeList}
- \texttt{addSwitchingCodeList}
- \texttt{addAwakenCodeList}
- \texttt{addRestoringCodeList}
- \texttt{addRestartingCodeList}
- \texttt{captureLocalVars}
5.3 Virtual OSGi Framework Migration Implementation

The operations’ names imply their usage as described above.
In order to illustrate how the bytecode transformer works a simple example of thread instrumentation will be presented. The example covers the case of instrumenting a thread method with no updates but with other instrumented nested methods in the call hierarchy.

**Example** Here, there is a service object which creates a WorkerThread instance that subclasses `java.lang.Thread`. This WorkerThread invokes some methods from an instance of a Helper class. The `run()` method of the WorkerThread looks like following:

```java
public String string;

public void run() {
    Helper h = new Helper();
    Helper h2 = new Helper();
    string = "this string";
    int i = 1;

    while (!isInterrupted()) {
        h.call(h2.call2(1), "2");
    }
}
```

As we see from the above code the run method’s instruction list has two invoke instructions (for sake of simplicity, we ignore `isInterrupted()`), one for `call()` and one for `call2()`. This means that the instrumentation mechanism should insert restoring block in the beginning of `run()` with two jumps just before the two method invocation instructions. It should also add capturing initiator code blocks before each nested method invocation and as these invocation are accessible for instrumentation, it should add capturing propagation code blocks, too. The instrumented code is presented below (some instructions are described in textual form for sake of readability):

```java
public String string;

public void run() {
    Helper h;
    Helper h2;
    int i;

    {restoring code block}{
        if Task is restoring{
            pop artificial program counter from context stack
            switch(program counter)
            1:
                goto _L10
            2:
                goto _L20
        }
    }
```
_L10:
{restoring code block}{}
// restore local variables
i = Context.popInt();
h2 = (Helper) Context.popObject();
h = (Helper) Context.popObject();
// restore fields
string = (String) Context.fields.popObject();
// restore stack
(Helper) Context.popObject(); // h
(Helper) Context.popObject(); // h2
Context.popInt(); // 1
if(context.stackEmpty())
{
   StateManager.restoreComplete(Thread.currentThread());
}
goto L1;
}

_L20:
{restoring code block}{}
// restore local variables
i = Context.popInt();
h2 = (Helper) Context.popObject();
h = (Helper) Context.popObject();
// restore fields
string = (String) Context.fields.popObject();
// restore stack
(Helper) Context.popObject();
Context.popInt();
(String) Context.popObject();
if(context.stackEmpty())
{
   StateManager.restoreComplete(Thread.currentThread());
}
goto _L2;
}
}
{restarting code block}{}
if Task is restarting
// restore only fields
string = (String) Context.fields.popObject();
context.clearContext();
}

h = new Helper();
h2 = new Helper();
string = "this string";
i = 1;

while (!isInterrupted()) {
    push h
    push h2
    push 1
    goto _L1

    _L1:
    {capturing code block}{
        if is Capturing
        // capture stack
        Context.pushInt();
        Context.pushObject();
        Context.pushObject();
        // capture fields
        Context.pushObject(string);
        // capture local variables
        Context.pushObject(h);
        Context.pushObject(h2);
        Context.pushInt(i);
        // store program counter
        Context.pushInt(1);
        return;
    }
    call2();
    {switching code block}{
        if is Switching
        JVM INSTR pop ;
        // capture stack
        Context.pushObject();
        // capture fields
        Context.pushObject(string);
        // capture local variables
        Context.pushObject(h);
        Context.pushObject(h2);
        Context.pushInt(i);
        // store program counter
        Context.pushInt(1);
        return;
    }
}

push "2"
goto _L2

_L2:
{capturing code block}{
    // capture stack
Context.pushObject();
Context.pushInt();
Context.pushObject();
// capture fields
Context.pushObject(string);
// capture local variables
    Context.pushObject(h);
    Context.pushObject(h2);
    Context.pushInt(i);
// store program counter
    Context.pushInt(2);
    return;
}
call();
{switching code block}{
    // capture fields
    Context.pushObject(string);
    // capture local variables
        Context.pushObject(h);
        Context.pushObject(h2);
        Context.pushInt(i);
    // store program counter
        Context.pushInt(2);
        return;
}
}

From the above instrumented code we see two restoring and one restarting block. They are in the beginning of the method body. The restoring block fetches the program counter from the Context object. Then, it switches on its value and jumps to the corresponding restoring block. At the end of each restoring block, the stacks of the Context are checked and if they are empty the restoration process is complete. In this case, theStateManager is notified to set the Task into running state again. The restoring code block will first restore the local variables of the method and the fields of the running instance. The local variables and fields have been stored in the corresponding capturing or switching code block. In the end, it will reestablish the stack state. We can also see two capturing code blocks before each nested method invocation. The capturing code blocks store the elements in the reverse order from the one they are restored (stack properties). Two switching code blocks (or capture propagation code blocks) appear too, just after each nested method invocation returns. These switching code blocks first capture the state of the stack. Notice here that not all of the stack operands are saved in the switching code blocks. The only operands that are saved are the operands that are on the operand stack just before the method invocation instruction, except for the ones consumed by the method invocation instructions. For example, in the case of call2 method invocation, the operand stack just before the call2 method invocation instruction has h,
5.3 Virtual OSGi Framework Migration Implementation

h2 and 1 operands from bottom to top. Operands h2 and 1 are consumed by call2 method invocation and are not saved. They are saved by a switching or capturing code block inside call2 as we will see below. In the case of call method invocation, the operands on the stack are h and "12" from top to bottom. These operands are consumed by the call method invocation and will not be saved here. They will be saved by the capturing and switching code blocks inside call body.

As mentioned above, the instrumentation does not stop here. Bodies of methods call and call2 should be instrumented, too. The instrumentation of call2 is illustrated below.

```java
public int call2(int i)
{
    {restoring code block}{
        if is restoring
            pop program counter from the Context stack
            // restore fields
            list = Context.fields.popObject();
            // restore stack
            Context.popObject();
            goto _L0;
    }

    push list;
    {capturing code block}{
        // save stack
        Context.pushObject();
        // save fields
        Context.fields.pushObject(list);
        // save program counter
        Context.pushInt(0);
        // save calling arguments and object reference
        Context.pushInt(i);
        Context.pushObject(this);
        return 0;
    }

    _L0:
    size();

    return 11;
}
```

The call2 method has a nested method invocation inside its body, which cannot be instrumented. So, we see that there is no switching code block after size method invocation. The interesting point is in the capturing code block. We can see that after storing the stack state, the fields and the artificial program counter, the arguments and the reference to the instance on which the method was called are pushed on the Context in reverse order, in order to be restored by the caller. This is done in the capturing blocks of all nested methods that can be instrumented.
If, for example, a replication or migration request was issued at the invocation of \texttt{call12} method, but after the capturing code block just before \texttt{call12}, the stack frame of method run just before the \texttt{call12} method invocation instruction would be like in figure 5.2. In the figure first the local variable slots and then the operand stack are shown.

![Figure 5.2: run() stack frame](image)

In figure 5.3 we can see the stack condition just before the capturing code block inside the \texttt{call12} method. In this figure another stack frame has been created for method \texttt{call12}.

![Figure 5.3: Stack before capturing](image)

The stack should be reestablished in exactly the same state as it is in the figure. Below follow the actions performed by the instrumented code and how the Context evolves in order to reestablish the stack. First of all, the replication or migration request will be noticed in the capturing code block of \texttt{call12} method. This will result in the following snapshot of the Context stack.

![Figure 5.4: Context stack after call2() capturing](image)
Then the capturing code will return control to `run()` method. There, the switching code block just after `call2()` method invocation will be executed. The first instruction is a JVM pop instruction. This is done in order to save everything that was on the stack before the method invocation, except for the operands associated with `call2()`. So, the result of `call2()` must be popped. After the execution of the code the `Context` stack will have evolved like the following figure.

![Figure 5.5: Context stack after run() switching](image)

After that, the return statement in the switching code block will terminate the thread. During the restoring state, after the migration has been finished, the restoring code block in the beginning of `run()` will be executed (in case of replication only the restarting code block will be executed). This will pop the program counter of `run` method from the top of the `Context`’s stack. The `goto` instruction will be executed and the control will be redirected at label L10 (see code above). The restoring code block there will pop local variables i, h2 and h from the `Context` stack with this order and will restore the thread’s local variables. Then, it will pop the `string` field’s value and restore it. The `string` field value will be popped from the fields’ stack. After that, the restoration of stack begins. The following elements on the `Context` stack were not pushed by `run` method’s switching code block, but from the `call2` method’s switching code. However, they will be popped from `run()`. h, h2 and i will be popped and pushed in this order on the run method’s operand frame. Now, the control flow will jump to `call2` method’s invocation instruction, with the help of the `goto` bytecode instruction. `call2` is ready to be executed with the correct operands on the stack. Apart from that, when `call2` returns, the stack will continue being in the correct state as the object reference of the next method invocation has already been pushed on the method’s stack frame. Then the invocation of `call2` method will take place, which will also execute its restoring code block. The restoring code block of `call2` pops the program counter from the `Context`’s stack, restores the value of the `list` field and pushes it on the stack. Then the stack frame of `call2` is ready for the invocation of `size()`. The jump executed by the corresponding `goto` instruction moves the control flow to the invocation of `size()`. Concluding, everything was restored correctly and the state is the same as before the migration request was issued.
Special cases

A special case in migration scenarios is when a service is being accessed by an external thread, at the time when a migration request is issued. This case needs special handling because the invocation started should be executed and return a correct result despite the migration request. For this reason, the root object of the service is wrapped inside a `ServiceWrapper` object. This wrapper registers each thread that is accessing the service when a method is invoked and de-registers it when the invocation is over. This registration is done in the `StateManager`. In this way, the `StateManager` knows at each time that a migration request can be issued if the service is in use. If such a case occurs, a mechanism called `temporary replication` is initiated. Each wrapper is associated with an `isLocked` flag. This flag is set when a migration request is issued. With the help of this flag, every new method invocation on the service will throw an exception from now on, as the service has been migrated. At the same time, the invocations that are in progress when the migration request is issued have their threads suspended. Instead of migrating the service, a replica at the new location is created and when the replication is complete the external threads are resumed. When the method invocations of these threads return the temporary replica is dropped and only the service in the migration location exists anymore. Similarly, when there is a replication request and the service is being accessed by an external thread the external thread(s) are suspended until the replication operation is complete. Unlike `temporary replication` any method invocations that may happen until replication is complete do not throw an exception because this replica should exist after replication too. The threads that access the service during a replication operation block until replication operation is complete. Once the new replica is created the threads are resumed and they execute like normal. This blocking is done with the use of the `isBlocked` flag associated with the wrapper, which is set once a replication operation is initiated.

Quantitative analysis

The quantitative analysis for the migration implementation requires very fine-grained measurements. First, of all it is clear that instrumenting and injecting code introduces time and space overhead. Since code is inserted for each `invoke-instruction` that occurs in the program, the space overhead is directly proportional to the total number of `safe points`, i.e. nested method invocations. The space overhead of the instrumentation is a function of the number of local variables in the scope of the instructions and the number of values that are on the operand stack executing the instruction.

What was more important for the system was not the minimum space overhead but the minimum time overhead during normal operation. Hence, the performance of a method invocation that encloses three nested methods was measured in order to see the average run-time overhead. The results are shown in the following table and figure (for 100000 invocations).

From the measurements it is shown that the run-time overhead does not exceed 10%.
### 5.3 Virtual OSGi Framework Migration Implementation

<table>
<thead>
<tr>
<th>Measurement #</th>
<th>Uninstrumented code</th>
<th>Instrumented code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>288199</td>
<td>300761</td>
</tr>
<tr>
<td>2</td>
<td>265978</td>
<td>289041</td>
</tr>
<tr>
<td>3</td>
<td>258727</td>
<td>274095</td>
</tr>
<tr>
<td>4</td>
<td>252825</td>
<td>271841</td>
</tr>
<tr>
<td>5</td>
<td>259411</td>
<td>272121</td>
</tr>
</tbody>
</table>

Table 5.2: Measurement of 100000 invocations of method

Figure 5.6: Measurements of 100000 invocations of a method
Chapter 6

Evaluation

As a sample real-world application a use-case of the Terracotta software was used: HTTP Session Replication in a multiple application server environment fronted by a load-balancer. As application server, Jetty 6.1.9 [17] was used. Jetty is an open-source, standards-based, full-featured web server implemented entirely in Java. It is released under the Apache 2.0 license and is therefore free for commercial use and distribution. Jetty and Virtual OSGi Framework replication libraries were deployed on a cluster of the Department of Computer Science, ETH Zürich, running the balance [5] load balancer.

6.1 HTTP Session replication

HTTP Session replication was selected as a usage and benchmarking scenario, because keeping sessions available across application servers in the event of an application server failure has long been an expensive and difficult problem to solve. The current trend in the application design has been to move away from storing application state in the session and towards what has been called a 'stateless' application design where the application state is stored in some external system like a database. But the term 'stateless' is not really valid in the above technique. State does not disappear but it is stripped from the application and moved to an external location. This can keep applications more flexible but it brings some undesirable side-effects. The first is performance: writing application state to a database makes that database a bottleneck so that scaling a cluster by adding application servers suffers from diminishing returns. The second is that the applications become more complicated, as the developer has the burden of getting acquainted with the APIs needed to externalize the application’s state. It is much much easier to put session relative data in the Session Java object. This the reason that this object exists. Session replication keeps the architecture of web applications unmodified, providing at the same time high availability and scalability.

6.2 Experimental environment

The testing software used during the experiments was Web Performance Suite 3.5 [28]. The Web Performance Suite is comprehensive and easy to use Web
performance testing software, which includes an analysis module (Analyzer) and a load testing module (Load Tester). It is the only performance testing software capable of handling most popular web-design technologies, without having to write complex code in a programming language. The testing procedure is straightforward: a browser is used to access the website in the same way as a real world user would. Web Performance Load Tester analyzes all the HTTP/S transactions, and automatically detects and configures embedded state and session variables, as well as page validation. The servlet deployed at each server whose performance was measured should include session-tracking into the tests. The servlet used was one used also by Web Performance to compare the performance of J2EE servers. The tests run made continuous requests to the server, by simulating load that varied between 1 and 10 users with random arrival and departure of users. Each request contained modification of a session’s attribute. The user scenario used represented long, involved operations on a website (e.g. placing an order and entering payment and shipping information). The duration of the test scenario was 3 minutes with 20 web pages with sizes 4x60k, 4x40k and 12x12k. As mentioned above, the web server used was Jetty 6.1.9, once instrumented with Terracotta replicated sessions and once with our replication implementation. The test was repeated in two different scenarios:

- 10 nodes: In the first scenario, 10 instances of replicated Jetty were running on ten different nodes of the cluster and the coordinator (or Terracotta server) was running on a separate node.

- 20 nodes: In the second scenario, 20 instances of replicated Jetty were running on twenty different nodes of the cluster and the coordinator (or Terracotta server) was running on a separate node.

The experiments were executed with the use of altruistic locking in the beginning. As expected, a great degradation in the performance of our system was observed, because of the reasons explained in the Distributed Synchronization section. After that, greedy locking was used where the performance was much better and the results are presented below.

The measurements for these two test scenarios are depicted in figures 6.1 and 6.2. The detailed data can be found in the Appendix A. The measurements show that our replication mechanism performs on average at least 5-10% better than Terracotta. This can be explained by the fact that our system does not use a single server for storing the state. The use of the coordinator is similar to the server of Terracotta, but it is more lightweight, as it does not store any object data. It acts as a plain sequencer and forwarder of updates and is responsible for coordinating thread concurrency between JVMs.
6.2 Experimental environment

Average Page Duration Jetty-Terracotta vs. Jetty-V-OSGi in 10 nodes cluster

Figure 6.1: 10 nodes measurements

Average Page Duration Jetty-Terracotta vs. Jetty-V-OSGi in 20 nodes cluster

Figure 6.2: 20 nodes measurements
Chapter 7

Replication through VM Support

During this thesis, an alternative replication implementation was elaborated, too. This was a native implementation and it was based on the JVM Tool Interface \[18\] and the Java Native Interface \[19\]. JVMTI is a new feature of the Java Virtual Machine (it first appeared on the JDK 5.0).

7.1 JVM Tool Interface

The JVM Tool Interface (JVMTI) is a new native programming interface for use by tools. It provides both a way to inspect the state and to control execution of applications running in the Java Virtual Machine. JVMTI supports the full breadth of tools that need access to JVM state, included but not limited to: profiling, debugging, monitoring, thread analysis and coverage analysis tools. JVMTI replaces the Java Virtual Machine Profiler Interface (JVMPI) \[16\] and the Java Virtual Debug Interface (JVMDI). JVMTI is a two-way interface. On the one hand, it allows registration of JVMTI clients, the agents, which are notified of interesting occurrences through JVMTI events. On the other hand, JVMTI can query and control the application through the use of JVMTI functions. The agents are running in the same process with the application and can be written in any native language that supports C language calling conventions and C or C++ definitions.

The Java Native Interface (JNI) is a programming framework that allows Java code running in the Java Virtual Machine to call and be called by native libraries. The JNI is used to write native methods to handle situations when an application cannot be written entirely in the Java programming language. In the context of JVMTI, the agents are the native libraries that should interact with the Java applications. From Java code perspective, methods natively implemented are declared with the native method modifier. The libraries implementing the native methods are shared libraries and are loaded in the same memory space with the Java application. JNI also defines how data types are represented in a native library. When a native method is called it is provided a pointer to the JNI environment. With this environment interface, it can call Java method, convert between C and Java data types and perform other JVM
specific operations such as creating Java threads and synchronizing on a Java monitor. A restriction with the JNI environment pointer that deserves additional attention is that the pointer is thread specific. It cannot be passed from one thread to another.

7.1.1 Class Redefinition

A shortcoming of the replication technique described in section 4.2 is that all the instrumentation of the application code should be done during loading of the application classes. Once the application classes are loaded, no additional instrumentation can be done. This means that every single instruction that may be executed by the application and could lead to a state change has to be instrumented. This can lead to significant space overhead, since the instrumentation adds instructions to the application. These instructions are added even in places, which may not be reached by any control flow branch during runtime. The problem is that the instrumentation is done before knowing which parts of the code the application will be executed. In other words, a pessimistic approach is used, where every possible point leading to state change should be instrumented.

JVMTI provides a class modification functionality. Instrumentation can be inserted in two ways:

- **Load-time instrumentation**
  When a class file is loaded by the VM, the raw bytes of the class file are sent for instrumentation to the agent. This mechanism provides efficient and complete access to one-time instrumentation.

- **Dynamic instrumentation**
  A class which is already loaded (and possibly even having some of its methods currently executing) is modified. Classes can be modified multiple times and can be returned to their original state. The mechanism allows instrumentation which changes during the course of execution.

Classes can be redefined at any point at runtime by providing the redefined bytecodes. The class redefinition requires that the agent has already acquired the corresponding capability. An original method version which is not equivalent to the new method version is called obsolete and is assigned a new method id. The new method version is used for new invocations. If a method has active stack frames, those active frames continue to run the bytecode of the original method version. These frames can also be popped by the agent, if the original method's functionality is no longer desired. The class redefinition does not cause any initialization. The values of static fields will remain as they were prior to the call. Instances of the redefined class are not affected either. Fields retain the previous values. A significant restriction of class redefinition is that it can change only method bodies, the constant pool and attributes. The redefinition must not add, remove or rename fields or methods, change the signature of methods, modifiers, or inheritance.

7.1.2 JVMTI Events

JVMTI and its predecessor, JVMPI, are event-based systems. The agent is informed for many events occurring in the application programs. To handle events,
a set of callback functions are declared and registered by the agent. For each event, the corresponding callback function is called. Arguments to the callback function provide additional information about the event. The callback function is called from within the application threads and the JVMTI implementation does not queue events in any way. All events are initially disabled by default and have to be explicitly enabled and disabled by the agent. Most events require that the agent has acquired the corresponding capability. The capabilities allow the agent to change the functionality available to JVMTI at the time where the VM starts. This includes the JVMTI functions that can be called, what events can be generated and what functionality these events and functions can provide. Frequently, the addition of a capability may incur a cost in the execution speed, start up time and/or memory footprint.

Events are delivered on the thread that caused the event. They are sent at the time they occur. The most important types of events offered by JVMTI are:

- Events for the life-cycle of the JVM
- Events for the life-cycle of classes
- Events for the life-cycle of threads
- Events for the life-cycle of objects
- Events for method calls
- Events for monitor contention
- Events for field access/modification
- Events for the garbage collector

The most important event type in the replication context, is the one for field access/modification. The agent can declare the fields that should be watched for access and/or modification. The main restriction of this mechanism is that it can be done only at the class level, leading to the generation of many events in cases of many instances of the class in the heap. Once the agent has registered the watched fields, a FieldAccess or FieldModification event is generated when the field specified by a class and a field id is accessed or modified. The corresponding events provide information about the thread that generated this event and the method and location where this event was generated. Field accesses and modifications from Java programming language code or from JNI code are watched, fields accessed or modified by other means are not watched.

### 7.1.3 Heap analysis

JVMTI provides functions used to analyze the heap. Functionality includes the ability to view the objects in the heap and to tag these objects. A tag is a value associated with an object. Tags are explicitly set by the agent. Objects which have not been tagged have a tag of zero. The tags are useful for information at instance and not class level.
7.2 Virtual OSGi Framework Native Replication

For the native implementation an agent was implemented that runs together with the application. The agent should be added to the library path. The agent requires the class name for the service object as an argument, to begin the instrumentation. As the agent is loaded, it acquires all the required capabilities needed for the instrumentation. Then, the callback functions for the events are registered with the JVMTI environment and the environment is informed about the types of events for which the agent should be notified. As mentioned in section 7.1.2, the overhead of acquiring a capability and adding event notification can hurt the application performance. Therefore, the agent should register for as few events as possible. The events required for the replication mechanism are first of all the field access and field modification events. The agent acquires also the capability to redefine classes, because some bytecode engineering is needed too.

7.2.1 Watched Fields and Object Tagging

Once the agent is loaded, it begins the instrumentation of the service. First of all, it registers the fields that are be watched and generate field access and modification events. In order to do this, the agent finds the class of the service object and iterates and recurses over the fields of the object’s class, notifying the JVMTI environment that these fields should be watched for access and modification. This recursion over the fields can be deep, consequently the number of watched fields can be very large. Since the field access/modification watch and the field access/modification events are on the class level, events will be generated for fields of all the classes’ instances, even if they are not referenced by the service object, i.e. they do not amount to state. Thereby, a mechanism to filter these events is needed. The filtering is achieved by the use of object tagging provided by the JVMTI environment. During the recursion over the fields, the agent fetches the values assigned at the fields and in case of reference type fields it tags the objects referenced. Consequently, every object referenced by the service will be tagged in the heap.

7.2.2 Class Redefinition

Once the agent has registered the fields that should be watched, the JVMTI environment will notify the agent for all the accesses and modifications of the fields. This means that no instrumentation of the application’s bytecode is needed to capture the changes in service state. However, this does not mean that no bytecode engineering is needed at all. There are three reasons for which bytecode instructions should be injected in the application’s code:

- **Updates on array elements**
  JVMTI offers notification for field access and modification. However, if a field of a class is an array, the elements of the array are not fields of the class. However, if the array referenced by the field amounts to state, the elements of the array amount to state too. Hence, every store operation on such an array should be intercepted. If the array has primitive type
elements, the instrumentation is complete by just intercepting these updates. The injected bytecode instructions will call a function of the agent, which will check the tag of the array object. If the object is tagged, the array amounts to state and the update will be stored and propagated. If not, the array operation will be ignored. In the case of an object array, the case is more complicated. The array store operation will be intercepted likewise, but the object referenced by the array’s element should be tagged too. Since it is referenced by an element of an array that amounts to state, the object referenced amounts to state too. Therefore, the object has to be tagged and the fields of its class should be watched for access/modification. This will be done recursively as above.

- **Thread synchronization and cooperation**
  JVMTI offers events about `MONITORENTER` and `MONITOREXIT` instructions. However, these events are generated only when a thread is attempting to enter a Java programming language monitor already acquired by another thread. Hence, these instructions should be intercepted, too. Bytecode instructions are injected which call an agent’s function which the object which is the owner of the monitor that is to be acquired. If this object is tagged, the cluster-wide lock should be acquired. This is done by skipping the original `MONITORENTER` instruction and calling the `distMonitorEnter` primitive used in the replication mechanism implemented in Java (see section 4.2.4). The `MONITOREXIT` instructions are handled likewise. Concerning distributed synchronization, JVMTI offers an event that is generated when a thread is about to wait on an object’s monitor. Therefore, distributed synchronization can be handled either by registering for this type of events and check for the tag of the monitor’s owner object or by intercepting the `wait` instructions. Either way, the `notify` instructions have to be intercepted, since no event of this type exists in JVMTI. After the check on the monitor owner’s tag, a call to the corresponding Java primitives (see section 4.2.4) will be done if the owner is tagged, i.e. it amounts to state.

- **Updates propagation**
  Finally, bytecode should be injected which will handle the update propagation (to the coordinator) in case of a service method exit. This is similar to the `EOT` primitive mentioned in section 3.4.

### 7.2.3 Access/Modification Callbacks

During the callbacks for field modification events, the owner object of the field is checked if it is tagged or not. If it is, the update is stored in the currently executing transaction (see section 3.4). When the transaction is over, the updates are propagated. These steps are illustrated in figure 7.1. As mentioned in section 4.2.3, the updates received by a node in the Virtual OSGi Framework are enqueued into a scheduler queue and are applied asynchronously. Therefore, when a field of an object that amounts to state is accessed, any possible pending updates should be applied immediately. With the help of the JVMTI field access events, when a field is accessed JVMTI notifies the agent about it. The agent checks if the owner object of the field is tagged. In case it is tagged, it checks if any updates for this field have been received but
88 Replication through VM Support

Figure 7.1: Field Modification Callback

not yet been applied, by making a call to the service’s Scheduler (see 4.2.3). These operations are illustrated in figure 7.2.

Figure 7.2: Field Access Callback

7.3 Performance Evaluation

The same measurements performed on the Java implementation of the Virtual OSGi Framework replication mechanism (see chapter 6) were performed on the native implementation too, so as to be able to compare the performance of the two different implementations. The results are illustrated in figure 7.3. The detailed data can be found in the Appendix B.

From the measurements performed a significant overhead of at least 35% was observed for the native implementation. This might seem contradicting but there are reasons for this overhead. First of all, as mentioned above, even the acquirement of a single type of capabilities can introduce significant overhead. With the generation of events this overhead will become much larger. If one considers that for this replication mechanism, two types of events and five types of capabilities were used, the overhead seems reasonable. A second argument
7.3 Performance Evaluation

Figure 7.3: 10 nodes measurements

for this overhead is that the events generated are on the class level, leading to the generation of many events, many of which may not amount to state. For example, registering for field access/modification events for a class that has many instances in the JVM heap will mean that all these objects in the heap will generate events. These events should be parsed and all of those not amounting to state add significant overhead to the application for no reason. Finally, the class redefinition mechanism provided by the JVMTI does not allow class schema change. This means that no new fields and methods can be added and the signature of the existing ones cannot be changed. Consequently, all the constructs used for replication, which in the Java implementation were stored in fields or passed as arguments to instrumented methods, had to be searched in hashtables in the native implementation. The search in a hashtable is much more expensive than a simple field access. To sum up, JVMTI (at its current version) was proved insufficient in the replication context.

Nevertheless, the native implementation had also some advantages. As mentioned above, it is not restricted to one-time instrumentation. A class can be redefined as many times as needed, even if it has already been loaded. Additionally, the space overhead by the instrumentation is minimal, thanks to the events provided by JVMTI. Finally, a class is instrumented in the native implementation only when really needed, because all the instrumentation is done at runtime. As a result, instrumentation needed by branches of the control flow never reached, will never be done.
Replication through VM Support
Chapter 8

Conclusions

During this thesis, the skeleton of the Virtual OSGi Framework’s replication and migration capabilities was built. This skeleton can be used by the upper layer of the framework which decides which services should be replicated and migrated and to which locations. The system built provides clear primitives to this layer in order to replicate and migrate the services easily and transparently. The first goal that was achieved in this thesis was to replicate applications in an efficient and transparent way. The replication mechanism needed dealing with Java bytecode at a very fine granular level. This was the interesting and at the same time difficult part of the project. The injected bytecode should capture every update happening at a service and do it in a lightweight way. Replication also involved extending Java synchronization with cluster-wide semantics. Dealing with thread coordination and mutual exclusion is tricky and time consuming in a single JVM. When these semantics are extended to be applied over multiple JVMs the situation becomes more complicated. The danger of deadlocks needs very careful design and implementation decisions. Moreover, the replication mechanism should not hurt the applications performance above a certain margin. This is needed in order to be able to argue that this system can be applicable in a real world scenario. A system with poor performance can never be applicable in reality. The evaluation of the system compared to existing implementations showed that its performance is good enough to be realistic and this was the greatest goal achieved.

Two different techniques were elaborated to inject this clustering behavior to the application. The first one was implement in Java and performed bytecode engineering at load time. This technique proved to be the most efficient one, having however some shortcomings. The first restriction was its one-time instrumentation property. A class could not be modified once it was loaded, therefore the technique should inherit worst-case scenarios properties. State capturing bytecode instructions were injected at every possible point of state change, even if these points were never reached during runtime. Therefore, the space overhead can be high. The second technique elaborated was implemented partially in Java and partially natively. This technique proved to add more overhead to the application performance. However, with the help of the Java VM Tool Interface, classes were redefined at any point during runtime, irregardless of whether they had already been loaded or not. No useless instrumentation was added, since the instrumentation was done only when really needed.
Conclusions

The second goal achieved was thread serialization. Java supports multi-threading and code mobility in the language level. However, mobility does not hold for threads. Thread's state is hidden inside the virtual machine and it cannot be accessed in its entirety using the Java API. Therefore, mechanisms to capture the thread's state at the application level had to be elaborated. The application's performance should again remain as close as possible to its plain execution. Threads are an inseparable part of the Java language and as a result migration, without thread serialization is unacceptable. The system's thread serialization mechanism was implemented again with load time bytecode instrumentation and shared the same difficulties with replication at the bytecode level. However, it was very interesting and gave a full understanding of a thread's internals and consequently of Java internals.

8.1 Future Work

As mentioned at the chapter of replication, two locking mechanism were used. The second one, greedy locking (see section 4.2.4), turned to be the most efficient one. However, it had the disadvantage of requiring $2 \times \text{RTT}$ time in order to transfer the lock state from one node (in case of leasing) to another. This policy cannot be optimal when locks are really contented frequently between nodes. A challenge would be to design some kind of heuristic to overcome this overhead. The lock could, for example, be leased to a node for a specific time period and after that period the lock would turn out to be shared by everyone again. These heuristics could also be applied depending on the lock's usage.

The symbolic code analysis could give an overview of the lock's usage frequency during runtime and the probability that it will be requested by many nodes.

A second proposal for future work could be to enhance the scheduler of the Active Object pattern with logic. The scheduler could have an overview of the transactions being executed and give priorities to some updates or even not apply some updates by predicting that another update will overwrite its effect. The scheduler could also have logic to deal with transaction serialization. It could keep snapshots about fields likely to become conflicting. This could be used to implement a rollback-like mechanism for conflicts. This would help the system switch from conflict detection and resolution to conflict avoidance which can in some cases turn out to be very efficient. The logic of the scheduler could also exploit the information received for every transaction when \texttt{BOT} primitive is called. The transaction could specify except for the fields accessed, the form of each access, i.e. \texttt{READ}, \texttt{WRITE}, \texttt{READ/WRITE}. In this way, the \texttt{getField} operation of the scheduler called during execution of \texttt{GETFIELD} instruction could be eliminated. The scheduler will know which fields will be read when the transaction begins and it will apply all of the pending updates then. Likewise, it can throw away pending field assignments, when it decides that the transaction ready to be executed will overwrite these values.

JVMTI can also be used for the implementation of a thread serialization mechanism with minimum bytecode engineering. JVMTI provides useful functions and events for thread state monitoring. Any thread can be suspended and resumed at any point in time. Additionally, all the information required about the stack is exported by the JVMTI environment. Stack frames are referenced by depth and the content of the local variables of each frame can be accessed.
and modified. The variables are identified by the depth of the frame and their slot number within that frame. The mapping of variables to slot numbers is obtained by retrieving the local variable table of each frame. Hence, the thread execution can be suspended, the stack trace can be retrieved and reestablished by setting the local variables to their old values with the help of the JVMTI functions.

As another proposal for future work, the JVMTI field access and modification events can also be very useful. These events are generated just before the access and just before the modification of a field, respectively. Different consistency models be evaluated by exploiting these notifications by the eventing mechanism. As an example, a two-phase commit (2PC) protocol can be used when a field modification event is encountered. At the time of a field modification, the nodes can communicate and decide whether an update should be applied or not. According to the decision, the new value of the field given by the event can be applied (commit) or skipped (abort).
Appendix A

Evaluation of the Java implementation
<table>
<thead>
<tr>
<th>Terracotta Average #</th>
<th>Min</th>
<th>Max</th>
<th>Virtual OSGi Framework Average #</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.073</td>
<td>1.073</td>
<td>1.073</td>
<td>0.827</td>
<td>0.802</td>
<td>0.852</td>
</tr>
<tr>
<td>0.731</td>
<td>0.386</td>
<td>1.049</td>
<td>0.504</td>
<td>0.331</td>
<td>0.678</td>
</tr>
<tr>
<td>0.506</td>
<td>0.181</td>
<td>0.971</td>
<td>0.422</td>
<td>0.169</td>
<td>0.676</td>
</tr>
<tr>
<td>0.364</td>
<td>0.177</td>
<td>0.937</td>
<td>0.378</td>
<td>0.166</td>
<td>0.801</td>
</tr>
<tr>
<td>0.393</td>
<td>0.187</td>
<td>0.958</td>
<td>0.340</td>
<td>0.166</td>
<td>0.534</td>
</tr>
<tr>
<td>0.518</td>
<td>0.181</td>
<td>0.968</td>
<td>0.415</td>
<td>0.169</td>
<td>0.692</td>
</tr>
<tr>
<td>0.496</td>
<td>0.182</td>
<td>0.966</td>
<td>0.266</td>
<td>0.168</td>
<td>0.488</td>
</tr>
<tr>
<td>0.342</td>
<td>0.181</td>
<td>0.574</td>
<td>0.296</td>
<td>0.168</td>
<td>0.489</td>
</tr>
<tr>
<td>0.322</td>
<td>0.175</td>
<td>0.558</td>
<td>0.283</td>
<td>0.169</td>
<td>0.489</td>
</tr>
<tr>
<td>0.337</td>
<td>0.176</td>
<td>0.497</td>
<td>0.381</td>
<td>0.168</td>
<td>0.493</td>
</tr>
<tr>
<td>0.388</td>
<td>0.178</td>
<td>0.691</td>
<td>0.330</td>
<td>0.168</td>
<td>0.486</td>
</tr>
<tr>
<td>0.403</td>
<td>0.176</td>
<td>0.726</td>
<td>0.301</td>
<td>0.168</td>
<td>0.488</td>
</tr>
<tr>
<td>0.301</td>
<td>0.178</td>
<td>0.508</td>
<td>0.294</td>
<td>0.164</td>
<td>0.484</td>
</tr>
<tr>
<td>0.289</td>
<td>0.176</td>
<td>0.511</td>
<td>0.247</td>
<td>0.165</td>
<td>0.487</td>
</tr>
<tr>
<td>0.374</td>
<td>0.176</td>
<td>0.509</td>
<td>0.324</td>
<td>0.166</td>
<td>0.483</td>
</tr>
<tr>
<td>0.380</td>
<td>0.179</td>
<td>0.506</td>
<td>0.327</td>
<td>0.167</td>
<td>0.488</td>
</tr>
<tr>
<td>0.377</td>
<td>0.179</td>
<td>0.629</td>
<td>0.289</td>
<td>0.168</td>
<td>0.487</td>
</tr>
<tr>
<td>0.319</td>
<td>0.176</td>
<td>0.500</td>
<td>0.304</td>
<td>0.167</td>
<td>0.500</td>
</tr>
<tr>
<td>0.316</td>
<td>0.179</td>
<td>0.500</td>
<td>0.321</td>
<td>0.168</td>
<td>0.491</td>
</tr>
<tr>
<td>0.353</td>
<td>0.179</td>
<td>0.502</td>
<td>0.298</td>
<td>0.168</td>
<td>0.486</td>
</tr>
<tr>
<td>0.369</td>
<td>0.178</td>
<td>0.506</td>
<td>0.327</td>
<td>0.167</td>
<td>0.497</td>
</tr>
<tr>
<td>0.279</td>
<td>0.179</td>
<td>0.521</td>
<td>0.270</td>
<td>0.166</td>
<td>0.503</td>
</tr>
<tr>
<td>0.325</td>
<td>0.177</td>
<td>0.551</td>
<td>0.303</td>
<td>0.165</td>
<td>0.495</td>
</tr>
<tr>
<td>0.346</td>
<td>0.176</td>
<td>0.504</td>
<td>0.328</td>
<td>0.164</td>
<td>0.500</td>
</tr>
<tr>
<td>0.352</td>
<td>0.226</td>
<td>0.496</td>
<td>0.166</td>
<td>0.163</td>
<td>0.168</td>
</tr>
<tr>
<td>0.262</td>
<td>0.224</td>
<td>0.338</td>
<td>0.167</td>
<td>0.166</td>
<td>0.168</td>
</tr>
<tr>
<td>0.195</td>
<td>0.176</td>
<td>0.228</td>
<td>0.164</td>
<td>0.164</td>
<td>0.165</td>
</tr>
<tr>
<td>0.182</td>
<td>0.180</td>
<td>0.186</td>
<td>0.269</td>
<td>0.164</td>
<td>0.481</td>
</tr>
<tr>
<td>0.396</td>
<td>0.180</td>
<td>0.507</td>
<td>0.167</td>
<td>0.167</td>
<td>0.167</td>
</tr>
<tr>
<td>0.391</td>
<td>0.337</td>
<td>0.493</td>
<td>0.246</td>
<td>0.167</td>
<td>0.325</td>
</tr>
<tr>
<td>0.231</td>
<td>0.176</td>
<td>0.341</td>
<td>0.166</td>
<td>0.166</td>
<td>0.167</td>
</tr>
<tr>
<td>0.228</td>
<td>0.227</td>
<td>0.230</td>
<td>0.164</td>
<td>0.164</td>
<td>0.165</td>
</tr>
<tr>
<td>0.179</td>
<td>0.176</td>
<td>0.183</td>
<td>0.164</td>
<td>0.164</td>
<td>0.165</td>
</tr>
<tr>
<td>0.391</td>
<td>0.176</td>
<td>0.503</td>
<td>0.325</td>
<td>0.167</td>
<td>0.483</td>
</tr>
<tr>
<td>0.288</td>
<td>0.180</td>
<td>0.343</td>
<td>0.246</td>
<td>0.167</td>
<td>0.325</td>
</tr>
<tr>
<td>0.285</td>
<td>0.180</td>
<td>0.494</td>
<td>0.166</td>
<td>0.166</td>
<td>0.167</td>
</tr>
<tr>
<td>0.274</td>
<td>0.214</td>
<td>0.335</td>
<td>0.163</td>
<td>0.162</td>
<td>0.165</td>
</tr>
<tr>
<td>0.216</td>
<td>0.187</td>
<td>0.229</td>
<td>0.166</td>
<td>0.164</td>
<td>0.168</td>
</tr>
<tr>
<td>0.181</td>
<td>0.181</td>
<td>0.182</td>
<td>0.171</td>
<td>0.171</td>
<td>0.171</td>
</tr>
</tbody>
</table>

Table A.1: 10 nodes cluster test case
<table>
<thead>
<tr>
<th>Terracotta</th>
<th>Virtual OSGi Framework</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Average #</strong></td>
<td><strong>Min</strong></td>
</tr>
<tr>
<td>1.024</td>
<td>1.003</td>
</tr>
<tr>
<td>0.703</td>
<td>0.347</td>
</tr>
<tr>
<td>0.483</td>
<td>0.180</td>
</tr>
<tr>
<td>0.805</td>
<td>0.180</td>
</tr>
<tr>
<td>0.585</td>
<td>0.404</td>
</tr>
<tr>
<td>0.525</td>
<td>0.249</td>
</tr>
<tr>
<td>0.376</td>
<td>0.229</td>
</tr>
<tr>
<td>0.180</td>
<td>0.443</td>
</tr>
<tr>
<td>0.439</td>
<td>0.228</td>
</tr>
<tr>
<td>0.531</td>
<td>0.343</td>
</tr>
<tr>
<td>0.375</td>
<td>0.188</td>
</tr>
<tr>
<td>0.306</td>
<td>0.183</td>
</tr>
<tr>
<td>0.338</td>
<td>0.180</td>
</tr>
<tr>
<td>0.412</td>
<td>0.182</td>
</tr>
<tr>
<td>0.460</td>
<td>0.340</td>
</tr>
<tr>
<td>0.342</td>
<td>0.183</td>
</tr>
<tr>
<td>0.339</td>
<td>0.181</td>
</tr>
<tr>
<td>0.307</td>
<td>0.176</td>
</tr>
<tr>
<td>0.424</td>
<td>0.185</td>
</tr>
<tr>
<td>0.413</td>
<td>0.336</td>
</tr>
<tr>
<td>0.340</td>
<td>0.179</td>
</tr>
<tr>
<td>0.285</td>
<td>0.178</td>
</tr>
<tr>
<td>0.417</td>
<td>0.177</td>
</tr>
<tr>
<td>0.414</td>
<td>0.336</td>
</tr>
<tr>
<td>0.219</td>
<td>0.179</td>
</tr>
<tr>
<td>0.191</td>
<td>0.179</td>
</tr>
<tr>
<td>0.362</td>
<td>0.181</td>
</tr>
<tr>
<td>0.335</td>
<td>0.335</td>
</tr>
<tr>
<td>0.341</td>
<td>0.189</td>
</tr>
<tr>
<td>0.388</td>
<td>0.333</td>
</tr>
<tr>
<td>0.182</td>
<td>0.181</td>
</tr>
<tr>
<td>0.392</td>
<td>0.181</td>
</tr>
<tr>
<td>0.284</td>
<td>0.183</td>
</tr>
<tr>
<td>0.538</td>
<td>0.177</td>
</tr>
<tr>
<td>0.257</td>
<td>0.180</td>
</tr>
<tr>
<td>0.234</td>
<td>0.172</td>
</tr>
<tr>
<td>0.388</td>
<td>0.177</td>
</tr>
<tr>
<td>0.288</td>
<td>0.181</td>
</tr>
<tr>
<td>0.363</td>
<td>0.363</td>
</tr>
<tr>
<td>0.183</td>
<td>0.183</td>
</tr>
</tbody>
</table>

Table A.2: 20 nodes cluster test case
Appendix B

Evaluation of the native implementation
### Evaluation of the native implementation

<table>
<thead>
<tr>
<th>Virtual OSGi Native</th>
<th>Average #</th>
<th>Min</th>
<th>Max</th>
<th>Virtual OSGi Java</th>
<th>Average #</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.331</td>
<td>0.827</td>
<td>1.331</td>
<td>0.852</td>
<td>0.450</td>
<td>0.331</td>
<td>0.678</td>
<td></td>
</tr>
<tr>
<td>0.450</td>
<td>0.504</td>
<td>0.450</td>
<td>0.678</td>
<td>0.444</td>
<td>0.422</td>
<td>0.615</td>
<td></td>
</tr>
<tr>
<td>0.444</td>
<td>0.213</td>
<td>0.581</td>
<td>0.676</td>
<td>0.408</td>
<td>0.360</td>
<td>0.801</td>
<td></td>
</tr>
<tr>
<td>0.408</td>
<td>0.624</td>
<td>0.624</td>
<td>0.534</td>
<td>0.378</td>
<td>0.340</td>
<td>0.534</td>
<td></td>
</tr>
<tr>
<td>0.378</td>
<td>0.621</td>
<td>0.621</td>
<td>0.592</td>
<td>0.320</td>
<td>0.415</td>
<td>0.692</td>
<td></td>
</tr>
<tr>
<td>0.320</td>
<td>0.617</td>
<td>0.617</td>
<td>0.488</td>
<td>0.534</td>
<td>0.266</td>
<td>0.488</td>
<td></td>
</tr>
<tr>
<td>0.534</td>
<td>0.213</td>
<td>0.595</td>
<td>0.489</td>
<td>0.374</td>
<td>0.296</td>
<td>0.489</td>
<td></td>
</tr>
<tr>
<td>0.374</td>
<td>0.229</td>
<td>0.614</td>
<td>0.489</td>
<td>0.357</td>
<td>0.283</td>
<td>0.493</td>
<td></td>
</tr>
<tr>
<td>0.357</td>
<td>0.213</td>
<td>0.604</td>
<td>0.493</td>
<td>0.447</td>
<td>0.381</td>
<td>0.493</td>
<td></td>
</tr>
<tr>
<td>0.447</td>
<td>0.213</td>
<td>0.618</td>
<td>0.486</td>
<td>0.374</td>
<td>0.330</td>
<td>0.486</td>
<td></td>
</tr>
<tr>
<td>0.374</td>
<td>0.243</td>
<td>0.687</td>
<td>0.488</td>
<td>0.494</td>
<td>0.301</td>
<td>0.488</td>
<td></td>
</tr>
<tr>
<td>0.494</td>
<td>0.213</td>
<td>0.596</td>
<td>0.484</td>
<td>0.417</td>
<td>0.294</td>
<td>0.484</td>
<td></td>
</tr>
<tr>
<td>0.417</td>
<td>0.213</td>
<td>0.618</td>
<td>0.487</td>
<td>0.352</td>
<td>0.247</td>
<td>0.487</td>
<td></td>
</tr>
<tr>
<td>0.352</td>
<td>0.213</td>
<td>0.618</td>
<td>0.483</td>
<td>0.415</td>
<td>0.324</td>
<td>0.483</td>
<td></td>
</tr>
<tr>
<td>0.415</td>
<td>0.218</td>
<td>0.596</td>
<td>0.488</td>
<td>0.454</td>
<td>0.327</td>
<td>0.488</td>
<td></td>
</tr>
<tr>
<td>0.454</td>
<td>0.229</td>
<td>0.611</td>
<td>0.487</td>
<td>0.455</td>
<td>0.289</td>
<td>0.487</td>
<td></td>
</tr>
<tr>
<td>0.455</td>
<td>0.235</td>
<td>0.659</td>
<td>0.500</td>
<td>0.520</td>
<td>0.304</td>
<td>0.500</td>
<td></td>
</tr>
<tr>
<td>0.520</td>
<td>0.229</td>
<td>0.630</td>
<td>0.491</td>
<td>0.358</td>
<td>0.321</td>
<td>0.491</td>
<td></td>
</tr>
<tr>
<td>0.358</td>
<td>0.229</td>
<td>0.628</td>
<td>0.486</td>
<td>0.398</td>
<td>0.298</td>
<td>0.486</td>
<td></td>
</tr>
<tr>
<td>0.398</td>
<td>0.228</td>
<td>0.677</td>
<td>0.497</td>
<td>0.394</td>
<td>0.327</td>
<td>0.497</td>
<td></td>
</tr>
<tr>
<td>0.394</td>
<td>0.229</td>
<td>0.631</td>
<td>0.503</td>
<td>0.446</td>
<td>0.270</td>
<td>0.503</td>
<td></td>
</tr>
<tr>
<td>0.446</td>
<td>0.239</td>
<td>0.718</td>
<td>0.495</td>
<td>0.462</td>
<td>0.303</td>
<td>0.495</td>
<td></td>
</tr>
<tr>
<td>0.462</td>
<td>0.228</td>
<td>0.643</td>
<td>0.500</td>
<td>0.318</td>
<td>0.328</td>
<td>0.500</td>
<td></td>
</tr>
<tr>
<td>0.318</td>
<td>0.229</td>
<td>0.626</td>
<td>0.491</td>
<td>0.379</td>
<td>0.166</td>
<td>0.491</td>
<td></td>
</tr>
<tr>
<td>0.379</td>
<td>0.239</td>
<td>0.625</td>
<td>0.491</td>
<td>0.349</td>
<td>0.167</td>
<td>0.491</td>
<td></td>
</tr>
<tr>
<td>0.349</td>
<td>0.229</td>
<td>0.453</td>
<td>0.500</td>
<td>0.275</td>
<td>0.164</td>
<td>0.491</td>
<td></td>
</tr>
<tr>
<td>0.275</td>
<td>0.229</td>
<td>0.633</td>
<td>0.500</td>
<td>0.363</td>
<td>0.269</td>
<td>0.491</td>
<td></td>
</tr>
<tr>
<td>0.363</td>
<td>0.228</td>
<td>0.629</td>
<td>0.500</td>
<td>0.346</td>
<td>0.167</td>
<td>0.500</td>
<td></td>
</tr>
<tr>
<td>0.346</td>
<td>0.237</td>
<td>0.436</td>
<td>0.325</td>
<td>0.303</td>
<td>0.246</td>
<td>0.325</td>
<td></td>
</tr>
<tr>
<td>0.303</td>
<td>0.229</td>
<td>0.325</td>
<td>0.325</td>
<td>0.452</td>
<td>0.166</td>
<td>0.325</td>
<td></td>
</tr>
<tr>
<td>0.452</td>
<td>0.240</td>
<td>0.433</td>
<td>0.325</td>
<td>0.338</td>
<td>0.164</td>
<td>0.325</td>
<td></td>
</tr>
<tr>
<td>0.338</td>
<td>0.229</td>
<td>0.604</td>
<td>0.325</td>
<td>0.327</td>
<td>0.164</td>
<td>0.325</td>
<td></td>
</tr>
<tr>
<td>0.327</td>
<td>0.229</td>
<td>0.620</td>
<td>0.325</td>
<td>0.380</td>
<td>0.325</td>
<td>0.325</td>
<td></td>
</tr>
<tr>
<td>0.380</td>
<td>0.240</td>
<td>0.443</td>
<td>0.325</td>
<td>0.294</td>
<td>0.246</td>
<td>0.325</td>
<td></td>
</tr>
<tr>
<td>0.294</td>
<td>0.228</td>
<td>0.605</td>
<td>0.325</td>
<td>0.416</td>
<td>0.166</td>
<td>0.325</td>
<td></td>
</tr>
<tr>
<td>0.416</td>
<td>0.238</td>
<td>0.434</td>
<td>0.325</td>
<td>0.337</td>
<td>0.163</td>
<td>0.325</td>
<td></td>
</tr>
<tr>
<td>0.337</td>
<td>0.228</td>
<td>0.242</td>
<td>0.171</td>
<td>0.233</td>
<td>0.166</td>
<td>0.171</td>
<td></td>
</tr>
<tr>
<td>0.233</td>
<td>0.239</td>
<td>0.171</td>
<td>0.171</td>
<td>0.239</td>
<td>0.164</td>
<td>0.171</td>
<td></td>
</tr>
<tr>
<td>0.239</td>
<td>0.239</td>
<td>0.171</td>
<td>0.171</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table B.1: 10 nodes cluster test case
Bibliography


