INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

# FT-OSGi: Fault-Tolerance extensions for the OSGi Service Platform

**Carlos Filipe Lopes Torrão**

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática e de Computadores

**Júri**

| | |
|---|---|
| Presidente: | Prof. Doutor José Carlos Martins Delgado |
| Orientador: | Prof. Doutor Luís Eduardo Teixeira Rodrigues |
| Vogal: | Prof. Doutor António Casimiro Ferreira da Costa |

**Outubro de 2009**

# Acknowledgments

I would like to start my acknowledgments by thanking my advisor, Prof. Luís Rodrigues, for all the guidance and comments provided in all this dissertation work. The support provided had a great influence to my experience and in the success of this dissertation completion. I want also to thank the given possibility to work in Distributed Systems Group (GSD) at INESC-ID, which had given to me a fruitful experience and insight of my research area during all my work.

I want also to thank Nuno Carvalho for all the assistance and insights provided in the development of this work, which were very useful and helpful.

I am grateful to all the GSD members of room 601 for the great work environment, and all the fruitful comments to my work. A special thanks to João Leitão, José Mocito and Liliana Rosa.

To end my acknowledgments, I want to express my sincere thanks to my friends and my family for all the support, belief and encouragement, which had a great importance in my personal commitment to this dissertation. Thank you very much!

This work was partially supported by the FCT project Pastramy (PTDC/EIA/72405/2006).

Lisboa, Outubro de 2009

Carlos Filipe Lopes Torrão

# Resumo

A plataforma OSGi define um sistema para o desenvolvimento de aplicações Java extensíveis e modulares. Este sistema também define uma arquitectura orientada a serviços (SOA) para aplicações, possibilitando um melhor isolamento e re-usabilidade de módulos de *software*. Visto que muitas das áreas de aplicação do OSGi têm requisitos de confiabilidade significativos, o principal objectivo desta dissertação é aumentar a confiabilidade das aplicações OSGi. Além disso, pretende-se também possibilitar a configuração de diferentes níveis de tolerância a faltas, de acordo com as necessidades de cada serviço OSGi. Deste modo, esta dissertação propõe uma série de extensões para a plataforma OSGi que permite a replicação de serviços OSGi, designada FT-OSGi. O FT-OSGi suporta várias estratégias de replicação, possibilitando uma diferente estratégia de replicação para cada serviço OSGi. Um protótipo da arquitectura foi concretizado e avaliado experimentalmente.

# Abstract

The OSGi Service Platform defines a framework for the deployment of extensible and downloadable Java applications. This framework also defines a service-oriented architecture (SOA) for applications, allowing a better module decoupling and reusability. Since many of the application areas for OSGi have significant dependability requirements, the main goal of this dissertation is to increase the dependability of OSGi applications. Furthermore, it aims at supporting different fault tolerance levels, according to the specific needs of each OSGi service. Thus, this dissertation proposes a set of extensions to the OSGi Service Platform that allow to replicate OSGi services, that we have named FT-OSGi. FT-OSGi supports multiple replication strategies, allowing to apply a different replication strategy to each OSGi service. FT-OSGi has been implemented and evaluated experimentally.

# Palavras Chave
# Keywords

## Palavras Chave

Confiabilidade

Tolerância a Faltas

Replicação

OSGi

Serviços

## Keywords

Dependability

Fault Tolerance

Replication

OSGi

Services

# Index

# List of Figures

# List of Tables

# Acronyms

**ACM** Association for Computing Machinery

**CORBA** Common Object Request Broker Architecture

**GCS** Group Communication System

**IEEE** Institute of Electrical and Electronics Engineers, Inc.

**IIOP** Internet Inter-ORB Protocol

**IP** Internet Protocol

**ISO** International Organization for Standardization

**LAN** Local Area Network

**MIT** Massachusetts Institute of Technology

**P2P** Peer to Peer

**QoS** Quality of Service

**RMI** Remote Method Invocation

**RPC** Remore Procedure Call

**SLP** Service Location Protocol

**SOA** Service-Oriented Architecture

**SOAP** Simple Object Access Protocol

**ORB** Object Request Broker

**OSGi** Open Services Gateway initiative

**TCP** Transmission Control Protocol

**UDP** User Datagram Protocol

**UPnP** Universal Plug and Play

**URI** Uniform Resource Identifier

**URL** Uniform Resource Locator

**WAN** Wide Area Network

**WLAN** Wireless Local Area Network

**WSDL** Web Services Description Language

**XML** eXtended Markup Language

# 1 Introduction

The OSGi Service Platform (OSGi Alliance 2007a) (Open Services Gateway initiative) defines a component-based platform for applications written in the Java programming language. The OSGi framework provides the primitives that allow applications to be constructed from small, reusable and collaborative components. The OSGi framework introduces a management unit, called *bundle*, that can be installed, updated, uninstalled, started or stopped without restarting the entire framework. There are also available higher level modules, called services, which augment and support a SOA (Bell 2008) approach for bundle interaction in the OSGi platform. These services, offer a better decoupling between different modules in OSGi. Therefore, they provide opportunities for increasing the dependability of OSGi based applications in a modular way. Hence, the OSGi framework allows fault-recovery to be applied at the service level and not at the level of the entire application. For instance, different fault-tolerance techniques may be applied to each individual service, depending of its characteristics. This dissertation addresses the problem of augmenting an OSGi framework with fault-tolerant mechanisms that exploit the modularity features of the framework.

The dissertation presents FT-OSGi, a set of fault tolerance extensions to the OSGi service platform. Our work has been inspired by previous work on fault-tolerant component systems such as Delta-4 (Powell 1994), FT-CORBA (Narasimhan, Moser, & Melliar-Smith 2002; Felber, Grabinato, & Guerraoui 1996; Baldoni & Marchetti 2003) and WS-Replication (Salas, Perez-Sorrosal, Pati & Jiménez-Peris 2006), among others. Our solution, however, targets problems that are specific for the OSGi platform. More precisely, the proposed solution enriches the OSGi platform with fault tolerance by means of replication (active and passive) in an almost transparent way to the clients, preserving the properties provided by the non fault-tolerant OSGi platform.

A prototype of FT-OSGi was implemented. This prototype leverages on existing tools, such as R-OSGi (Rellermeyer, Alonso, & Roscoe 2007) (a service that supports remote accesses

to OSGi services) and the Appia group communication toolkit (Miranda, Pinto, & Rodrigues 2001) (for replica coordination). The dissertation also presents an experimental evaluation of the fault-tolerance capabilities, that measures the overhead induced by the replication mechanisms offered by the FT-OSGi extensions.

## 1.1 Motivation

The Java language, by itself, provides little support for the modular deployment and maintenance of complex applications. The deployment unit in Java is a Java ARchive (JAR) file which has the Java classes that will be deployed. However, Java does not provide a standard way to manage these JAR files, such install, uninstall, start and stop them in runtime. The problem of this unmanageability for large applications is known as "Jar Hell", which is inspired in a similar problem with windows operating systems, called "DLL Hell". Kaegi and Deugo (2008) present a solution to this problem, based on the use of a Java modularity technology called OSGi (OSGi Alliance 2007a). OSGi has been adopted by several organizations as a step forward to improve the modularity of Java applications, including Eclipse, Nokia, Motorola, BMW, Spring, among others. The OSGi Service Platform was also developed with several applications' areas in mind, including domotics, automotive electronics, mobile computing and health electronics. Many of these application areas of OSGi have availability and reliability requirements. For instance, in domotic applications, reliability issues have been reported as one of the main impairments to user satisfaction (Kaila, Mikkonen, Vainio, & Vanhala 2008). Therefore, it is of utmost importance the design of fault-tolerance support for those applications.

## 1.2 Contributions

The dissertation addresses the problem of increasing the dependability of OSGi based systems. It proposes an architecture, called FT-OSGi, that offers modular fault-tolerance for OSGi applications. FT-OSGi has the following properties: increases the availability and reliability of OSGi services; provides different levels of the availability and reliability desirable to each OSGi service, according to the specific characteristics and requirements of each individual service; and, at last, achieves a solution with an acceptable performance.

## 1.3 Results

The dissertation has the following results:

- A running prototype of the FT-OSGi architecture based on OSGi (OSGi Alliance 2007a), R-OSGi (Rellermeyer, Alonso, & Roscoe 2007), Appia (Miranda, Pinto, & Rodrigues 2001), jGCS (Carvalho, Pereira, & Rodrigues 2006) and a number of components developed to fulfill the FT-OSGi goals.
- A experimental evaluation of the FT-OSGi prototype.

## 1.4 Research History

The dissertation work and research was done in the Distributed Systems Group (GSD) at INESC-ID institution (Portugal), during the 2008/2009 scholar year. In this period I expanded my knowledge and experience thanks to the fruitful discussions and comments by the GSD team. In particular, the collaboration with Nuno Carvalho in the development of the FT-OSGi prototype was of great importance, since it allow a better integration between the jGCS and Appia technologies with the FT-OSGi extensions prototype. The material provided by Jan S. Rellermeyer (R-OSGi developer) was also a good step forward for the evaluation of the FT-OSGi prototype.

The work developed has been published as a short-paper in (Torrão, Carvalho, & Rodrigues 2009a) and, it will appear as a full-paper in (Torrão, Carvalho, & Rodrigues 2009b). The FT-OSGi source code has been made available in sourceforge[1].

## 1.5 Dissertation Structure

The remaining of the dissertation is organized as follows. Chapter 2 presents all the background related with our work, including a survey on the main fault tolerance techniques, a description of a fault tolerant system for CORBA, a brief description of OSGi, and some fault-tolerance solutions for this framework. Chapter 3 presents the FT-OSGi, the fault tolerance

[1] http://sourceforge.net/projects/ft-osgi

extensions for the OSGi Service Platform. Chapter 4 presents the evaluation of the performance overhead induced by the FT-OSGi extensions in replicated services. At last, Chapter 5 concludes the dissertation.

# 2 Related Work

This chapter reviews the related work that is relevant to the project. It starts by surveying the fundamental concepts associated with dependable computing, including the main techniques to achieve fault-tolerance. Afterwards presents a fault tolerant system for CORBA applications and another for Web Services. Then it provides a brief introduction to the OSGi architecture. Finally, it summarizes some previous work that addresses the specific problem of increasing the dependability of OSGi systems.

## 2.1 Fault Tolerance

Fault tolerance can be shortly defined as a technique to increase the dependability of a system. This section starts by defining dependability, its attributes, threats, and means. Fault tolerance is one of the means for achieving dependability; as the names implies, it consists in tolerating faults, which are the first threat for dependability. After that, two main classes of faults are presented, classified according to their determinism. The deterministic faults are called Bohrbugs, and the nondeterministic ones are called Heisenbugs. Afterwards, the two main techniques used in replication are introduced, namely: active replication and passive replication. These techniques are important because fault tolerance is only achievable through redundancy, and replication fits in providing that. Afterwards, the main differences between hardware and software failures are identified, and are also described the reasons for the focus on software fault tolerance in our work. Subsequently, some techniques for software fault tolerance are presented, which are divided in two types: single-version and multi-version. Then, some significant dimensions of fault tolerant systems are introduced. A categorization of applications considering their fault tolerance strength is also provided. Technologies available to increase the fault tolerance levels and some case-studies are also surveyed. Finally, some fault injection techniques that can be used for benchmarking fault tolerant systems are referred.

### 2.1.1 Dependability

Computing systems are characterized by four properties: functionality, performance, cost and dependability. Following Avizienis et al. (Avizienis, Laprie, & Randell 2001), the dependability can be described as the ability of the system to deliver service that can justifiably be trusted, and it is characterized by the following attributes:

- **Availability:** Reflects the probability of a system operating correctly and being available to perform its functions at the considered instant of time, i.e., readiness for correct service.
- **Reliability:** Reflects the probability of a system operating correctly without any interruption in a complete interval of time, i.e., continuity of correct service. In opposition with the availability attribute, the reliability implies no interruptions between a certain period of time. For instance, a system can be high-available with fast interruptions, i.e., if the interruptions are a really small portion of time a system can be high-available, while still unreliable.
- **Safety:** Defines the absence of catastrophic consequences on the user(s) and the environment, i.e., the fail-safe system capability.
- **Confidentiality:** Defines the absence of unauthorized disclosure of information.
- **Maintainability:** Reflects the ability to undergo repairs and modifications.

To conduct a dependability evaluation and comparison of several systems is necessary to quantify the dependability attributes involved. Therefore, it is necessary to define the following metrics.

**Mean Time to Repair (MTTR)** is the average time necessary to repair the system after a failure (Benzekri & Puigjaner 1992).

**Mean Time between Failures (MTBF)** is the expected time (on average) that the system will operate until occurring a failure.

The reliability attribute is proportional to the Mean Time Between Failures (MTBF) (Gray 1986), and the availability can be quantified by the following formula:

$$\textbf{Availability} = \frac{MTBF}{MTBF + MTTR} \tag{2.1}$$

The dependability of a system can be compromised by three threats, namely:

- **Fault**: adjudged or hypothesized cause of an error. There are two types of faults: active and dormant; a fault is active when produces an error, otherwise it is dormant;
- **Error**: part of the system state that may cause a subsequent failure;
- **Failure**: occurs when an error reaches the service interface and alters the service.

A dependable computing system can be developed by using a combination of the following four complementary techniques: fault prevention, fault tolerance, fault removal and fault forecasting. All of these techniques are means to achieve system dependability.

#### 2.1.1.1 Fault Prevention

Fault prevention is related with the quality control employed during the design and conception of software and hardware. Operational physical faults may be prevented by shielding, radiation hardening, etc. Interaction faults may prevented by training, rigorous procedures for maintenance, "foolproof" packages, among others. Malicious faults may be prevented by firewalls and similar defenses.

#### 2.1.1.2 Fault Tolerance

Fault tolerance has the purpose of maintaining the delivery of correct service in the presence of active faults. Usually, fault tolerance relies on error detection followed by error recovery. Error detection can be classified in two classes: concurrent error detection (during the service delivery) and preemptive error detection (while service delivery is suspended). The recovery has the purpose of transforming the system state with errors and possibly faults into a system state without detected errors. For that purpose, the recovery consists in error handling and fault handling. Error handling is responsible for eliminating the errors from the system state; fault handling prevents the faults that have been located from being activated again. Errors can be eliminated using two main approaches: rollback (return to previous saved state without errors detected) and roll-forward (creation of a new state). Fault handling involves four steps: fault diagnosis, fault isolation, system reconfiguration and system reinitialization. It is crucial that

the mechanisms implemented by the fault tolerant system are also protected against faults that can affect them. Hence, fault tolerance has a recursive nature.

#### 2.1.1.3 Fault Removal

Fault removal consists of excluding the discovered faults from the system. It is present during the development phase and operational life of a system. During the development phase of a system, it requires three steps: verification, diagnosis and correction. Verification techniques can be categorized in two types: static and dynamic. Static verification can be performed without executing the system; on the other hand, dynamic verification is during the execution of the system. During the operational life of a system, fault removal may be implemented using corrective or preventive maintenance. Corrective maintenance aims at removing faults that produced one or more errors and have been reported. Preventive maintenance aims at removing faults before they might cause errors during normal operation.

#### 2.1.1.4 Fault Forecasting

Fault forecasting performs an evaluation of the system behavior based on fault occurrence or activation. The evaluation has two aspects:

- **Qualitative** or **ordinal** evaluation: aims to identify, classify, rank the failures or the event combinations that would lead to system failures;
- **Quantitative** or **probabilistic** evaluation: aims to evaluate in terms of probabilities the satisfaction degree of the dependability attributes.

The main concern of our work is increase the availability and reliability of software modules. That can be achieved using the techniques described above, but some faults are impossible to identify/correct and only the fault tolerance technique can solve that situations. Obviously, is desirable to combine all the techniques with each other to have an even better system dependability.

### 2.1.2 Fault Determinism

Faults can be characterized according to its determinism in two distinct classes (Gray 1986): Bohrbugs (deterministic faults) and Heisenbugs (nondeterministic faults).

Bohrbugs (the name is inspired on the Bohr atom) are faults usually easy to repeat, identify and correct. The repetitions of those kinds of faults are easy, because they normally depend only in the input gave and the current state of the system. Therefore, a specific Bohrbug that has happened for a particular input and system state, it will happen again and again for the same input and system state. With the right tools (for instance, debuggers) the identification and the correction of this kind of bug is usually easy.

On the other hand, the Heisenbugs (the name is inspired on the Heisenberg Uncertainty Principle in Physics) are faults really hard to repeat, identify and correct. This kind of faults can depend on a variety of characteristics, like internal clocks, threads synchronization, switch of threads, etc. This kind of characteristics is very hard to control, therefore the repetition of a specific Heisenbug is very hard to do. Consequently, the identification and correction of Heisenbugs are also very hard to do, and even the presence of a tool (debugger), with that purpose, can perturb enough the system to make the Heisenbug disappear on that condition. Gray and Siewiorek (1991) consider the name Heisenbug attributed to a transient fault, and they suggest that most software faults in production systems are transient.

### 2.1.3 Replication Techniques

Fault tolerance in a system requires some form of redundancy (Nelson 1990), and replication is one of the main techniques to achieve it. Typically, a replicated component is a concurrent component, as multiple clients may attempt to interact with different replicas at the same time. It is then of paramount importance to define precisely what is the correct behavior of the replicated component.

One of the most intuitive behaviors consists in requiring the replicated component to behave like a single non-replicated component. This allows replication to be transparent to applications that have been designed to operate with the non-replicated component. Such component respect the property of *linearizability* (Herlihy & Wing 1990), which sometimes is called as one-copy equivalence.

To ensure linearizability in this case, the subsequent properties must be fulfilled:

- **Order**: Given invocations $op(arg)$ by client $p_i$ and $op'(arg')$ by client $p_j$ on replicated server $x$, if two different replicas handle both invocations, they handle them in the same order.

- **Atomicity**: Given invocation $op(arg)$ by client $p_i$, on replicated server $x$, if one replica of $x$ handles the invocation, then every correct (non-crashed) replica of $x$ also handles $op(arg)$.

There are two main replication techniques that are able to ensure linearizability (Guerraoui & Schiper 1997): passive and active replication.

In passive replication, also known as primary-backup, one replica, called *primary*, is responsible for processing and respond to all invocations from the clients. The remaining replicas, called *backups*, do not process direct invocations from the client but, instead, interact exclusively with the primary. The purpose of the backups is to store the state changes that occur in the primary replica after each invocation. Furthermore, if the primary fails, one of the backup replicas will be selected (using some leader election algorithm previously agreed among all replicas) to play the role of new primary replica.

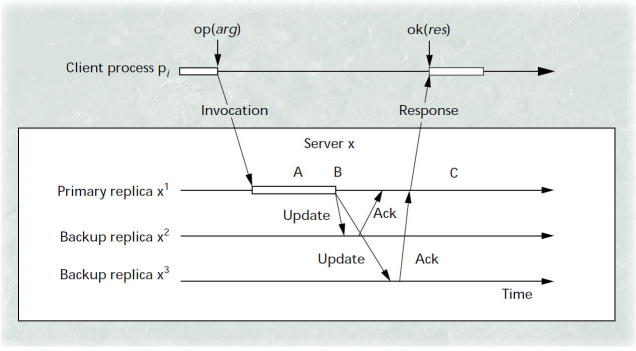



Figure 2.1: Example of a passive replication interaction

Figure 2.1[1] illustrated the processing of one invocation from a client and assuming that no failures occur. The steps are detailed below:

[1]Picture taken from (Guerraoui & Schiper 1997).

1. The client process $p_i$ sends $op(arg)$ to the primary replica ($x^1$) together with a unique invocation identifier, *invocationID*.

2. The primary replica ($x^1$) processes (invokes) $op(arg)$ and obtains the response *res*. Then, it captures the updated state (*stateupdate*) and sends it to the backups in an update message (*invocationID*, *res*, *stateupdate*). As a result, the backups update their state and return an acknowledgment to the primary replica.

3. When the primary replica receives the acknowledgments from all correct (non-crashed) backups, it returns the response *res* to the client process $p_i$.

Both properties of linearizability are fulfilled as follows: the order is ensured by the primary replica, and the reception of the *stateupdate* message by all backup replicas ensures the atomicity property. If the primary replica fails before responding, this replication technique requires the client process to re-issue the request.

In the active replication, also called the state-machine approach, all replicas play the same role thus there is no centralized control. In this case, all replicas are required to receive requests, process them and respond to the client. In order to satisfy the linearizability property, request need to be disseminated total-order-multicast (also known as atomic multicast). This primitive ensures the properties required to ensure linearizability: order and atomicity. This replication technique has the limitation that the operations processed by the replicas need to be deterministic (thus the name, state-machine).

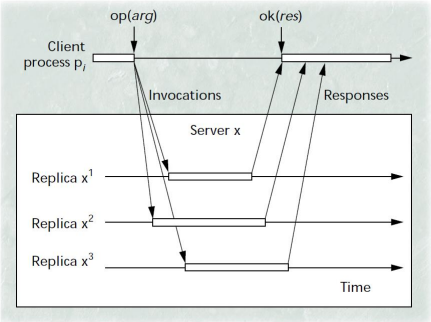



Figure 2.2: Example of an active replication interaction

Figure 2.2[2] illustrates a run of active replication technique for one invocation from a client and assuming no failures in replicas. The steps are detailed below:

[2]Picture taken from (Guerraoui & Schiper 1997).

1. The invocation $op(arg)$ is atomically broadcast to all the replicas of $x$.

2. Each replica processes the invocation, updates its own state, and returns the response to client process $p_i$.

3. The client process $p_i$ waits until it receives the first response or a majority of identical responses (depending if the client want to test the correctness of the replicas responses)

### 2.1.4 Hardware / Software Failures

The failures may be categorized in two distinct types: hardware and software.

#### 2.1.4.1 Hardware Failures

The high failure rate in the beginning of the hardware component life is higher because of something described as Infant-Mortality, which is usually caused by manufacturing problems like poor soldering, leaking capacitor, etc. However, in general these kinds of hardware components do not pass the tests executed by the manufactory. After this life phase, during the useful life of the hardware component, the fail rate decreases to a low rate, and for that reason is possible to have a better belief in the hardware component during this phase. In the last phase (end of life) there is a continuous increase of the fail rate, leading, sooner or later, to a failure of the hardware component. This increase happens because hardware suffers from degradation through its life time, which does not happen with software. The fail rate can be greatly reduced, almost to zero, with the introduction of replicated hardware modules in the system, preferably within the hardware expected life time to ensure a smaller fail rate.

#### 2.1.4.2 Software Failures

It is kind of an agreement that software is more complex than hardware, and this complexity is increasing day after day. Therefore, this complexity is the cause of the increasing rate of software faults introduced during the development phase of the software system. Most of the faults are corrected during the testing phase of the project, but some unpredictable faults (usually Heisenbugs) can still cause failures to the system, and that is the main reason of why the

focus in our work will be related with software faults. Besides this, the software fault tolerant solutions are usually a cheaper way to provide a better dependability for applications.

### 2.1.5 Software Fault Tolerance

The key to providing high availability is to modularize the system so that modules are the unit of failure and replacement. Additionally, the combination of modularity and redundancy is the key to providing continuous service even if some components fail (Gray 1986). Subsequently it will be presented the two main redundancy approaches related to software modules: single-version and multi-version.

### 2.1.6 Single-Version Software Fault Tolerance Techniques

Single-version fault tolerance is based on the use of redundancy applied to a single version of a software module to detect and recover from faults. Considering this type of fault tolerance, it will be introduced two important techniques to achieve fault tolerance: checkpointing and process pairs.

#### 2.1.6.1 Checkpoint

As mentioned before, most of the software faults remaining after development and tests are Heisenbugs, which are unanticipated. They appear, do the damage and then apparently just go away, leaving no obvious reason for their activation. Therefore, one solution for these Heisenbugs is the restore of the failed module and retry the same operation. Furthermore, a restart or rollback has the advantages of being independent of the damage caused by a fault, and general enough that it can be used at multiple levels in a system (Anderson & Lee 1981).

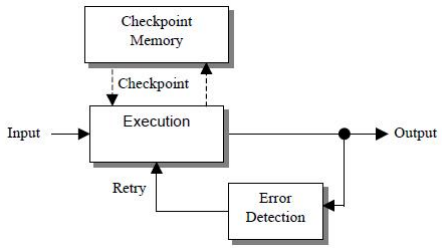



Figure 2.3: Representation of Checkpoint and Restart Technique

In Figure 2.3[3] is possible to observe a generic representation of a module using checkpoint and restart technique. The idea is simple, the module does a checkpoint of its state (for instance, during the execution, at fixed intervals of time) and it has an input and an output. If an error is detected (for example, wrong output) the module recover its previously checkpointed state and retries the same input. Notice that if the cause of failure was a Heiseinbug, the module in the retry will (usually) work properly. If not, probably the module is experiencing a Bohrbug, which is usually easy to identify and correct (Gray 1986).

#### 2.1.6.2 Process Pairs

A process pair uses the same software module running on separate processors (Pradhan 1996) and uses the checkpoint and restart technique, described above, for the recovery. The processors are labeled as primary and secondary (or backup).

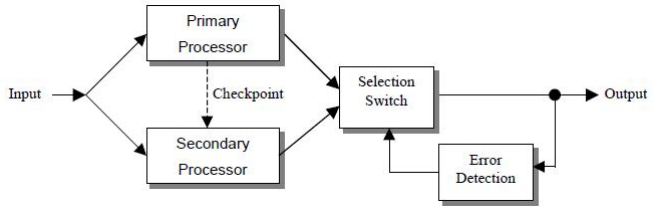



Figure 2.4: Representation of Process Pairs Technique

In Figure 2.4[4] is possible to observe a generic representation of this technique. The primary processor is actively processing the input and creating the output, at the same time it checkpoints its state and sends it to the secondary processor. When an error is detected, the secondary processor loads the last checkpointed state from the primary processor and it takes its role, being now the primary processor. The faulty processor takes the secondary role when becomes ready again for normal operation. The key advantage of this technique is the availability of the module, which continues uninterrupted after the incident of a failure in the system. This technique has a main limitation: the module can only have deterministic operations on it.

[3]Picture taken from (Wilfredo 2000).
[4]Picture taken from (Wilfredo 2000).

### 2.1.7 Multi-Version Software Fault Tolerance Techniques

Multi-version fault tolerance is based on the use of redundancy applied to two or more different versions of a software module to detect and recover from faults. These different versions can be executed in sequence or in parallel.

The motivation for the use of multiple versions for the same module is the expectation that modules built differently (different designers, different algorithms, different design tools, etc.) should fail differently (Avizienis & Chen 1977). Hence, if one version fails on a particular input, it is expected that other version it will be able to provide a correct output. Concluding, the multi-version can tolerate Bohrbugs and Heisenbugs but the single-version (described above) can only tolerate Heisenbugs.

This approach has some techniques to achieve fault tolerance, the two most important techniques are: recovery blocks and N-version programming.

#### 2.1.7.1 Recovery Blocks

The recovery blocks technique (Randell 1975; Randell & Xu 1995) combines the basics of checkpoint and restart approach with multiple versions of a software module. The checkpoints are created before a version executes, this is necessary to recover the state if the version fails.

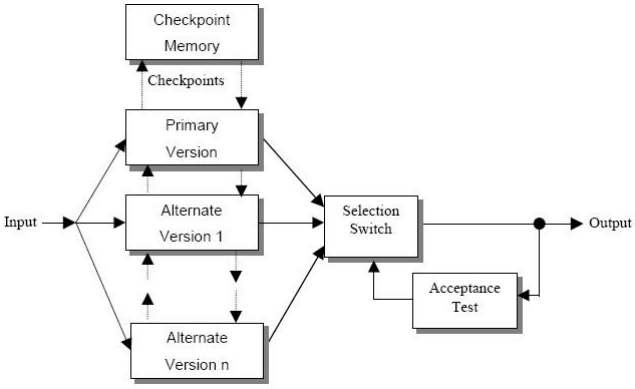



Figure 2.5: Recovery Blocks Model

This model is defined by a primary version and one or more alternate versions, which is possible to see in Figure 2.5[5]. The primary version will be executed successfully most of the

[5]Picture taken from (Wilfredo 2000).

time, but in case of failure in the acceptance test, a different version is tried, until the acceptance test passes or all versions were tried. The acceptance test does not need to be based only in output; it can be implemented by various embedded checks to increase the effectiveness of the error detection.

#### 2.1.7.2 N-Version Programming

The N-version programming technique (Avizienis 1995) is designed to achieve a decision of output correctness from multiple module versions (see Figure 2.6[6]).

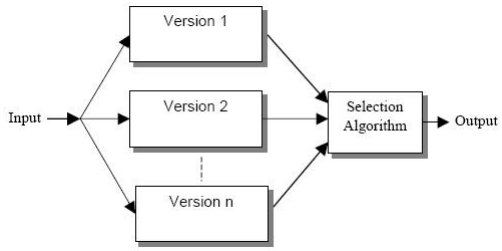



Figure 2.6: N-Version Programming Model

That decision is accomplished by a selection algorithm (usually a voter) to select the correct output from all the outputs of each version. This aspect is the main difference between this technique and the recovery blocks technique, which requires an application dependent acceptance test. The execution of the versions can be sequential or in parallel, but the sequential execution may need the use of checkpoints to reload the state before a different version is executed.

### 2.1.8 Dimensions of Fault Tolerance

The demands for fault tolerance in the applications software are a reality. Therefore, to fulfill those demands, there are some technologies possible to integrate with applications already developed or in development. It is possible to divide in two major dimensions the requirements of the applications: availability and data consistency. Ideally, these two dimensions should be as high as possible in any application with a need of fault tolerance. However, in reality, achieving that kind of perfection requires introduction of an undesirable amount of overhead

[6]Picture taken from (Wilfredo 2000).

to the application performance. Consequently, there is a prioritization of which dimension the application needs the most.

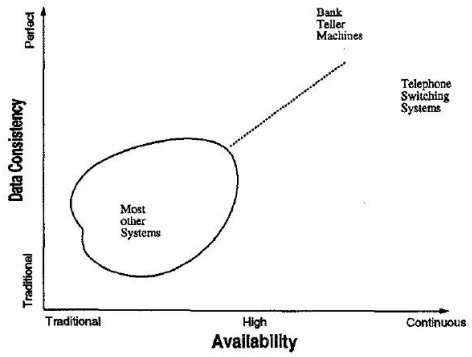



Figure 2.7: Dimensions of Fault Tolerance

Figure 2.7[7] shows a graph of the dimensions magnitude present in some systems. For example, the telephone systems prefer a continuous availability instead of a perfect data consistency, because it is not a huge problem if in a conversation of five minutes, one second of the conversation is lost, but it is a huge problem if the telephone systems are unavailable during five minutes. As an opposite example, the bank systems prefer a perfect data consistency instead of a continuous availability, because when the clients do a transaction they need to be sure of the transaction consistency instead of having the system always available but without that sure. Further in this section, it will be presented five software fault tolerance levels possible to classify any application, and how is it possible to achieve a specific level on any application simply by integrating some of the technologies available.

### 2.1.9 Software Fault Tolerance Levels

Before introducing the software fault tolerance application levels, it will be presented the usual model of applications based in client-server architecture.

Figure 2.8[8] presents a view of the components of an application. The application is running within an Operating/Database System, which is represented by a process (compiled code) with two kinds of data:

[7]Picture taken from (Huang & Kintala 1993).
[8]Picture taken from (Huang & Kintala 1993).

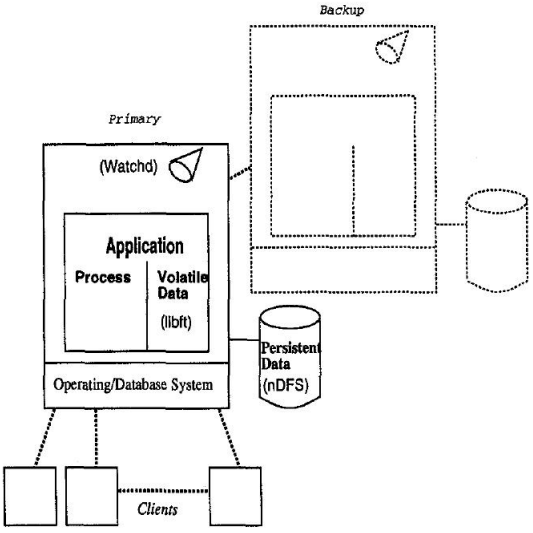



Figure 2.8: Example of an application in client-server architecture

- **Volatile data**: the variables, structures, pointers and all the bytes in the static and dynamic memory segments of the process;
- **Persistent data**: the files/information typically stored into a hard drive or database.

The interaction with the application is made by the clients.

Based on the major dimensions (availability and data consistency) presented above in this section, Huang and Kintala (1993) with their experience in AT&T decided to define five software fault tolerance levels for the applications.

#### 2.1.9.1 Level 0 - No fault tolerance in the application software

This level is defined by an application without any kind of fault tolerance. When the application hangs or crash, it has to be manually restarted and the internal state (volatile data) starts from the initial state. Besides that, it is likely that application leaves the persistent data in an incorrect or inconsistent state.

#### 2.1.9.2 Level 1 - Automatic detection and restart

This level is similar to the previous one, but the detection and the restart of the application are automatic. However, it has the same issues of Level 0 related with the internal state (volatile

data) and the persistent data. This level ensures a higher availability in comparison with Level 0.

#### 2.1.9.3 Level 2 - Periodic checkpointing, logging and recovery of the internal state

This level has the same characteristics of Level 1, with a difference in the volatile data consistency. In this level, the internal state of the application is periodically checkpointed, i.e., the volatile data is saved and the messages to the server are logged. After a failure is detected, the application is restarted with the last saved internal state (volatile data) and the logged messages are reprocessed to get the state of the application right before the occurred failure. The application availability and volatile data consistency in this level are higher than Level 1.

#### 2.1.9.4 Level 3 - Persistent data recovery

This level has the same characteristics of Level 2, with a difference in the persistent data consistency. The persistent data of the application is replicated through backup disks. And in case of application failure, the backup brings the persistent data close to the application state before the occurred failure. The data consistency in this level is higher than Level 2.

#### 2.1.9.5 Level 4 - Continuous operation without any interruption

This level of fault tolerance in software guarantees the highest degree of availability and data consistency. This is provided, for example, using replicated processing of the application on "hot" spare hardware. Multicast message, voting and other mechanisms must be used to maintain consistency and concurrency control.

### 2.1.10 Technologies and Experience

Huang and Kintala (1993) developed three reusable components possible to integrate with any application, these components are projected to increase the fault tolerance levels (defined previously) of applications.

#### 2.1.10.1 Watchd

Watchd is a watchdog daemon process that runs on a single machine or on a network of machines. This component is responsible for the detection of hangs or crashes of the application, and in that case, it will restart the application. It has two methods to detect the hangs. The first method sends null messages to the local application process using IPC (Inter Process Communication), and then checks the return value. The second method asks the application process to send heartbeat message periodically to the watchd. The watchd cannot distinguish between hung processes or very slow processes. After detecting the hang or crash of the application, watchd restart and recover the application at the initial internal state or the last state before the hang/crash, depending if the watchd is used in combination with Libft or not. The watchd can also recover an application to another backup node, in case the primary node has crashed. Another feature of the watchd is the capability of watching and recovering itself in case of its own failure. An application integrated with the watchd component can guarantee the Level 1 of the software fault tolerance levels.

#### 2.1.10.2 Libft

Libft is a user-level library of C functions that can be used in application programs to specify and checkpointing critical data, recover the checkpointed data, log events, locate and reconnect a server, do exception handling, do N-version programming, and use recovery blocks techniques. The checkpointing mechanism used by this library minimizes the overhead by saving only critical data and avoiding data-structure traversals. This idea is analogous to the Recovery Box concept in Sprite (Baker & Sullivan 1992). Watchd and Libft, when combined in an application, can guarantee the Level 2 of the software fault tolerance levels.

#### 2.1.10.3 nDFS

nDFS is a multi-dimensional file system based on 3DFS (Fowler, Huang, Korn, & Rao 1993) and provides facilities for replication of critical persistent data. Speed, robustness and replication transparency are the primary design goals of nDFS. An application using Watchd, Libft and nDFS can guarantee the Level 3 of the software fault tolerance levels.

The telecommunication network management products in AT&T has been enhanced using

these technologies. The experience with those AT&T products reveals that these technologies are indeed economical and effective means to increase the level of fault tolerance in application software. The performance overhead induced by these technologies varies from 0.1% to 14%, depending on which technologies are being used.

### 2.1.11 Fault Injection Techniques

Fault injection techniques are useful to test and evaluate the fault tolerance capabilities of a target system. These techniques can be divided, depending on the target system, in the two kinds: Hardware and Software. Following Hsueh et al. (Hsueh, Tsai, & Iyer 1997), software fault injections can be done at different stages: during compile-time or during runtime. Compile-time injection changes the program source code during the compilation, introducing faulty instructions with the intention of simulating faults during the execution of the respective program. This type of injection does not require any additional software running during the execution of the program, which is an advantage in comparison with the runtime injection, as it minimizes the perturbation to the system. On the other hand, the runtime injection has the benefit of being able to inject faults as the workload program runs. Furthermore, runtime injection may use three different mechanisms to trigger fault injections:

- **Time-out**: uses a timer to trigger the injection of faults to the program. It is a good option to introduce unpredictable fault effects.
- **Exception/trap**: it is based on the use of exceptions/traps to transfer the control to the fault injector. Unlike the time-out, this mechanism can inject faults when occur certain events or conditions.
- **Code insertion**: the target program suffers code insertion, during runtime, to inject faults to the program. This mechanism allows fault injection to take place before particular instructions.

## 2.2 Fault Tolerance in CORBA

In this section is presented a fault tolerance system for CORBA applications, called Eternal. This system had a large influence in the conception of the Fault-Tolerant CORBA standard (Ob-

ject Management Group 2001). At first this section presents a small description of CORBA, and afterwards an overview of the Eternal system. This overview of the Eternal system is included here because CORBA and OSGi systems have some similarities. Therefore, the techniques used by Eternal can solve some of the problems that can arise when providing fault tolerance to OSGi applications.

### 2.2.1 CORBA

The Common Object Request Broker Architecture (Object Management Group 2004) (CORBA) is a standard defined by the Object Management Group (OMG), that defines how objects executing in different nodes of a distributed system may make remote invocations among them. The interface of CORBA objects is specified in an Interface Definition Language (IDL); the client of a remote object only needs to be aware of the IDL interface, and not of the specific details of the object implementation.

The main component of CORBA model is the Object Request Broker (ORB), which acts as intermediary in the communication between a client object and a server object, shielding them from differences in programming language, platforms, and physical locations. Communication among clients and servers uses a standardized TCP/IP-based Internet Inter-ORB Protocol (IIOP).

### 2.2.2 Eternal

The Eternal system (Narasimhan 1999; Narasimhan, Moser, & Melliar-Smith 2002) is a component-based framework that provides transparent fault tolerance for CORBA applications. In this way, the application programmer does not need to be concerned with fault tolerant issues during the application development. The Eternal system provides fault tolerance by replicating CORBA objects. Therefore, each CORBA object is, in fact, implemented by several replicas, which guarantees a higher availability. The use of several replicas for a single object requires a strong replica consistency, which raises four concerns to achieve that consistency in a transparent way. The concerns are the following: ordering of operations, duplicate operations, recovery, and multithreading.

To maintain the replicas of the same object in a consistent state, these replicas need to

receive the operations in the same order. Eternal achieves this by using a reliable totally-ordered multicast protocol.

Since every replica of an object receives the same operation, therefore each replica will produce a response. Hence, this requires Eternal to remove the duplicated responses, because it is not expected that an object (even if replicated) produces more than one response for each invocation.

When a replica fails (and then is recovered) or a new replica is activated, Eternal needs to ensure, before the replica starts to operate, that replica has the same state of the other replicas of the object, which are already operational and with the same state. To achieve that, Eternal retrieves the state of the other replicas and applies it to the new or recovered replica.

Multithreaded objects can guide their replicas to an inconsistent state. Therefore, Eternal has mechanisms to overcome this problem.

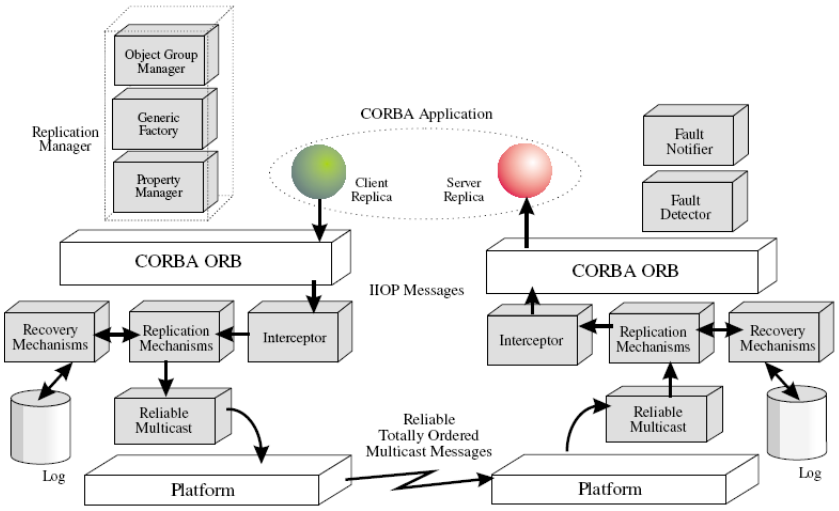



Figure 2.9: Architectural overview of the Eternal system

Figure 2.9[9] illustrates the components architecture of the Eternal system. The Replication Manager component, which is divided in three components, is responsible for replicating each object and distribute the replicas across the system, according the requirements specified by the user. The Property Manager component gives to the user the option of defining the fault tolerance properties to be used, such as the replication style, the consistency style, the initial and the minimum number of replicas, and the time interval for each state checkpointing. The

[9]Picture taken from (Narasimhan, Moser, & Melliar-Smith 2002).

Generic Factory component provides the functionality for the creation and deletion of object replicas. And, at last, the Object Group Manager component allows users to have direct control over the replicated objects.

The Fault Detector component is capable to detect host, process and object faults. To achieve this, each object inherits a Monitorable interface, which can provide a way to check the object status. Furthermore, the user can define properties like the monitoring style (pull or push) and the time interval to check the object status. In addition, the Fault Detector when detect a fault communicate that occurrence to the Fault Notifier. The Fault Notifier receives reports of faults and filters them to remove duplicate reports, and distribute the fault reports to all interested parties. One of those interested parties is the Replication Manager, which can start the appropriate recovery actions on receiving a fault report from the Fault Notifier.

The Interceptor component in the Eternal system attaches itself to each CORBA object at runtime, which provides a way to adapt the behavior of the object as desired. This component employs the library interpositioning hooks found on Unix and Windows NT. Therefore, the connections are transparently converted into connections to the Replication Mechanisms component.

Eternal provides three different types of replication: active, cold passive and warm passive. Therefore, the Replication Mechanisms component performs different operations for each type of replication. The active and passive replication mechanisms were already explained earlier in this chapter, and this component follows the same basis. The difference of the warm passive and the cold passive replication is related with the state transfer phase from the primary replica. The warm passive replication maintains synchronized all the backup replicas, but the cold passive replication does not load the backup replicas, instead just retrieves and stores in a log the state of the primary replica.

The Recovery Mechanisms component is responsible to recover, when needed, the three kinds of state present in every replicated CORBA object: application state, ORB state (maintained by the ORB) and infrastructure state (maintained by the Eternal). To enable the capture and recover of the application state is necessary that the CORBA object inherits a Checkpointable interface that contains methods to retrieve (*get_state()*) and assign (*set_state()*) the state for that object. Additionally, the Recovery Mechanisms log all new messages arriving during the time of state's assignment to a replica.

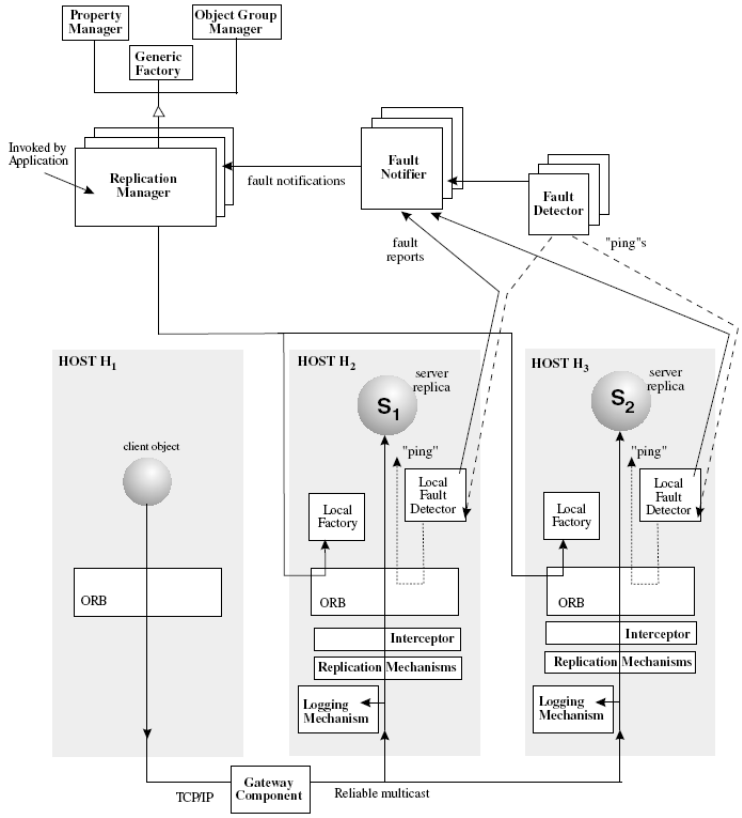



Figure 2.10: Interaction of Eternal components

Figure 2.10[10] shows the interaction of all the Eternal components described previously with a replicated object S, which has two replicas: S1 and S2.

An important limitation present in Eternal and also in other fault tolerant systems is the need for deterministic operations in their replicated objects, when using the active replication strategy.

## 2.3 Fault Tolerance in Web Services

The Web Service architecture is designed to provide remote services over a network, supporting interoperability between services deployed on heterogeneous environments. In critical applications, some of these web services, require high dependability. The WS-Replication (Salas, Perez-Sorrosal, Pati & Jiménez-Peris 2006) infrastructure applies the design principles introduced in the previous sections to offer replication in the Web Service architecture. It allows

[10]Picture taken from (Narasimhan, Moser, & Melliar-Smith 2002).

clients to access replicated web services. Replica consistency is ensured using a group communication web service, that has been adapted to execute on top of a web service compliant protocol (SOAP). The infrastructure supports three different replication strategies: active replication, semi-active replication, and passive replication. The active and passive replication follows the approach described before in Section 2.1. The semi-active replication allows the usage of non-deterministic code. It uses an approach inspired by the leader-follower technique (Powell 1994) where non-deterministic actions are executed by a leader replica. Afterwards, the leader sends the resulting state to the other replicas.

From an architectural perspective, the WS-Replication infrastructure is divided in two main components: a web service replication component and a reliable multicast component. The replication component is responsible for the web service replication and contains a web service deployer, a proxy generator, and a web service dispatcher. The web service deployer simplifies the deployment of replicated web services from a central location. The proxy generator is responsible to create the proxy responsible to intercept and replicate the invocations from a client. The dispatcher delivers the requests to the reliable multicast component, and delivers the replies to the proxy. The reliable multicast component is responsible to provide the group communication based on SOAP.

WS-Replication was subject to several improvements during the first prototype development. These improvements were able to improve the performance overhead induced by replication to an acceptable level.

## 2.4 OSGi

The OSGi Alliance was founded in March 1999 and it is the responsible for the creation and continuous progress of the OSGi Service Platform specification. The last specification of this service is the Release 4 (Version 4.1) (OSGi Alliance 2007a), which was released in April 2007. The description of the OSGi Framework, during this section, is based on that release.

The OSGi Framework forms the core of the OSGi Service Platform, which supports the deployment of extensible and downloadable applications, known as *bundles*. The OSGi devices can download and install OSGi bundles, and remove them when they are no longer required. The framework is responsible for the management of the bundles in a dynamic and scalable

way. One of the main advantages of the OSGi framework is the support for the bundle "hot deployment", i.e., the support to install, update, uninstall, start or stop of a bundle while the framework is running. At the time of writing of this dissertation, it is possible to find several OSGi framework implementations of the OSGi specification, such as Apache Felix (Apache Foundation 2009), Eclipse Equinox (Eclipse Foundation 2009) and Knopflerfish (Knopflerfish Project 2009).

### 2.4.1 Architecture

The OSGi Framework architecture and functionality is divided in the following major layers, as depicted in Figure 2.11[11]: Security Layer, Module Layer, Life Cycle Layer and Service Layer. There are some dependencies among these layers. The Module Layer can be used without the Life Cycle Layer and Service Layer. Additionally, Life Cycle Layer can be used without Service Layer. However, the Service Layer requires all the other layers.

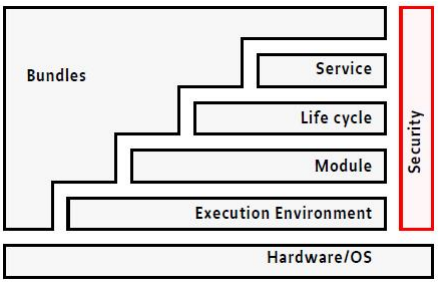



Figure 2.11: OSGi Framework Layers

#### 2.4.1.1 Security Layer

The Security Layer is based on Java 2 security architecture, by adding a number of constraints and defining some issues left open by the standard Java specification, which are necessary for the proper execution of the OSGi Framework.

#### 2.4.1.2 Module Layer

The Module Layer is responsible for the modularization model for Java. A bundle is the unit of deployment in the OSGi, which consists in a Java ARchive (JAR) file that contains a

[11]Picture taken from (OSGi Alliance 2007a).

manifest, described below, and some arrangement of Java class files, native code and associated resources. The manifest of a bundle contains information about dependencies on other resources (for instance, other bundles) and other information of how the Framework installs and activates a bundle. The modularization model defines strict rules for package sharing among distinct bundles, i.e., in which manner a bundle can export and/or import Java packages to/of another bundle.

#### 2.4.1.3 Life Cycle Layer

The Life Cycle Layer provides the API for the life cycle of bundles, which defines how bundles are installed, updated, uninstalled, started and stopped. Furthermore, it supplies a comprehensive event API to allow a management bundle to control the operations of the service platform. The life cycle of a bundle passes through the following possible states, illustrated in Figure 2.12[12]:

**INSTALLED:** the bundle was successfully installed;

**RESOLVED:** the classes imported by the bundle are available. In this state the bundle is ready to start or stop;

**STARTING:** the bundle is being started and it will become active when its activation policy allows it;

**ACTIVE:** the bundle was successfully activated and it is running;

**STOPPING:** the bundle is being stopped;

**UNINSTALLED:** the bundle was uninstalled.

#### 2.4.1.4 Service Layer

The Service Layer is based on a publish, find and bind model. A service is defined by a public Java interface, which is decoupled from its implementation, and bundles can register services (publish), search for them (find), or receive notifications when their registration state changes

[12]Picture taken from (OSGi Alliance 2007a).

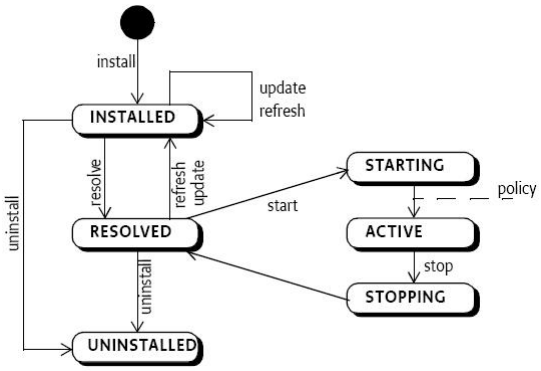



Figure 2.12: OSGi Bundle State Life Cycle

(when bound). Besides, a service runs within a bundle and this bundle is responsible to register the services owned in the OSGi Framework service registry, which maintains the information to let other bundles find and bind the services registered. Furthermore, the dependencies between the bundle owning a service and the bundles using it are managed by the OSGi Framework. Then, when a bundle is uninstalled, the OSGi Framework unregisters all the services owned by that bundle.

This layer provides the higher abstraction level achievable by the bundles, which gives a simple, dynamic, concise, and consistent programming model for bundle developers.

#### 2.4.1.5 Layer Interactions

Figure 2.13[13] shows the interaction between the OSGi layers (described above) and a bundle. The bundles are capable of register, unregister, get and unget OSGi services. The Life Cycle layer can start and stop a bundle, and also install and uninstall bundles in the Module layer. Besides, the Life Cycle layer can also manage the Service layer, for instance, when a bundle with registered services is unnistalled, then its services registered in the Service layer are also unregistered.

[13]Picture taken from (OSGi Alliance 2007a).

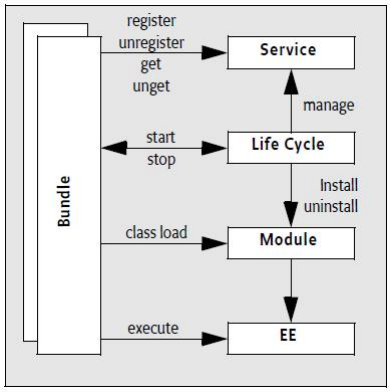



Figure 2.13: Interactions between the layers

## 2.5 Fault Tolerance in OSGi

OSGi Framework Release 4 does not address the problem of providing fault tolerance support for services. However, OSGi is being applied in systems with dependability requirements, including systems with high availability and reliability requirements. In the following paragraphs some works that address the problem of adding dependability features to OSGi based systems will be described, namely solutions for OSGi-based residential gateways (Thomsen 2006; Ahn, Oh, & Sung 2006) and a proposal for a dependable distributed OSGi (Maragkos 2008; Matos & Sousa 2008).

### 2.5.1 Replicated OSGi Gateways

Thomsen (2006) presents a solution to eliminate the single point of failure of OSGi-based residential gateways, which are responsible for home automation and the communication with all the home devices (through several physical interfaces, such as power lines, Ethernet, ZigBee). To achieve that, Thomsen creates sub-gateways for each type of network, allowing the creation of autonomous islands in case of the main gateway failure. Therefore, the system can still operate without the main gateway, even if with some limitations, which provides a form of graceful degradation. The system relies on the use of a passive replication based technique (also known as primary-backup), where the primary replica is the main gateway, and the sub-gateways are the backup replicas. To accomplish replication, Thomsen discussed the three following issues: What, How and When to replicate.

To maintain the sub-gateways always replicated is necessary to replicate the bundles (exe-

cutable code, internal state and persistent data) of the main gateway.

A bundle is composed by the executable code, internal state and persistent data. Each has a different way to reach replication. Since the executable code of a bundle is a JAR file, the replication is done by copy the JAR file. The internal state is replicated based on which position is the CPU processing of the code, and CPU registers and memory contents. The persistent data can be done based on the copy of the files/information stored in a persistent storage.

The executable code is replicated when the bundle suffers an update. The remaining data is replicated periodically or when happens a modification.

### 2.5.2 Proxy-based OSGi Framework

In a similar context, but with focus in the services provided through OSGi Framework, Heejune Ahn et al. (Ahn, Oh, & Sung 2006) presents a proxy-based solution, which provides features to monitor, detect faults, recover and isolate a failed service from other service. Consequently, this solution adds four components to the OSGi Framework: proxy, policy manager, dispatcher and monitor. A proxy is constructed for each service instance, with the purpose of controlling all the calls to that service. The monitor is responsible for the state checking of each service. Finally, the dispatcher decides and routes the service call to the best implementation available with the help of the policy manager. In this work, Heejune Ahn et al. only provide fault tolerance to a stateless service, therefore, the service internal state and persistent data are not recovered.

### 2.5.3 Dependable Distributed OSGi Environment

Nowadays, companies are using more and more services available in the Internet, like Amazon Web Services. These companies (costumers of services) expect that these services are provided with a determined Service Level Agreement. Therefore, the service providers need to guarantee to their costumers a strong isolation of their resources/services to give the illusion to a costumer that all the resources/services are available only to that costumer, when in fact, they are being shared. Matos and Sousa (2008) in their work provide an OSGi-based architecture and core services to fulfill that isolation with a concern in dependability aspects, which are a requirement of the service providers and costumers. Summarily, the goals are:

- Extend the OSGi Platform to be able to safely run multiple customers;
- Ability to migrate customers between nodes;
- Ability to measure resource usage of each customer;
- Ability to enforce Service Level Agreements requirements based on business policies.

Three architectures are presented. The first runs a JVM (Java Virtual Machine) for each OSGi Framework running and this is controlled by the Instance Manager, which is the external entity running also in a JVM (see Figure 2.14[14]). The problems of this architecture are the overhead caused by the multiple JVMs, the difficulty task of the Instance Manager and the lack of a "direct" method to communicate through instances.

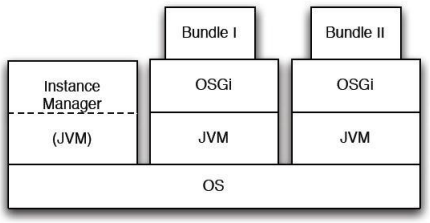



Figure 2.14: Architecture with multiple OSGi instances on different JVMs

A way to overcome the problems listed above is presented in the second architecture, which only uses one JVM to all the instances (see Figure 2.15[15]).

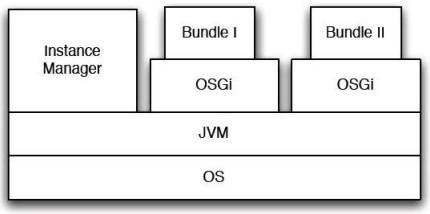



Figure 2.15: Architecture with multiple OSGi instances with only one JVM

Nevertheless, to provide a platform dynamic and modular, it makes sense to place the Instance Manager on top of the OSGi Framework with an OSGi environment on it, possible to have multiple instances (see Figure 2.16[16]). A good feature of this solution is the possibility

[14]Picture taken from (Matos & Sousa 2008).
[15]Picture taken from (Matos & Sousa 2008).
[16]Picture taken from (Matos & Sousa 2008).

to run bundles in the lower OSGi Framework which can communicate with the bundles in the multiple OSGi environments on top of the Instance Manager.

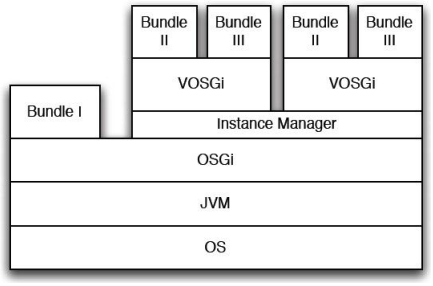



Figure 2.16: Architecture with multiple OSGi instances inside an OSGi environment

Matos and Sousa also describe three modules to this architecture: monitoring module, migration module and autonomic module. The monitoring module is responsible to measure the resource usage of each running instance and the overall resource availability. The migration module is responsible to migrate the virtual OSGi instances from one node to another node. The OSGi specification enforces that the Framework state shall be persistent across Framework reboots. Therefore, this makes the migration of the virtual OSGi Framework simple to do, but to be successful it is also required the migration of the bundles (including their state) in that OSGi Framework, which their state are not specified as persistent in the OSGi specification. Besides this, it is required a Storage Area Network or a distributed file system through nodes. For stateless bundles the solution is already solved, but stateful bundles requires a more complicated approach, which is not provided in their work, delaying this matter to future work. Finally, the autonomic module is responsible to enforce the business policies defined by the administrator.

## Summary

This chapter presented the related work relevant to our work, which will be described in the next chapter. This chapter, started with a survey of the fundamental concepts associated with dependable computing, with focus in the main techniques to achieve fault-tolerance. As examples of fault-tolerant systems, have been presented a fault-tolerant system for CORBA applications and another for Web Services. Then, a brief introduction to the OSGi architecture was made. Finally, some previous work addressing the specific problem of increasing the dependability of OSGi systems were described.

# 3 FT-OSGi

This chapter presents FT-OSGi, a set of extensions to the OSGi platform to improve the reliability and availability of OSGi applications. This chapter describes the services provided by such extensions and how they are implemented, detailing several components of FT-OSGi and how these components interact with each other in the different replication strategies.

## 3.1 Provided Services

The FT-OSGi provides fault tolerance to OSGi applications at the service layer. This is achieved by replicating OSGi services in a group of servers. To access the services, the client application communicates with the group of servers in an almost transparent way. The services can be stateless or stateful. If the service is stateful, the state management must be supported by the application programmer. In order to maintain the replicated state, the service must implement two methods: one for exporting its state (`Object getState()`) and another for updating its state (`void setState(Object state)`). The FT-OSGi extensions support three types of replication: active, eager-passive and lazy-passive. The strategy used for replication of an OSGi service is chosen at configuration time, and different services using different replication strategies can coexist in the same FT-OSGi domain.

In runtime, replication is supported by group communication. Each replicated service may use a different group communication channel or, for efficiency, share a group with other replicated services. For instance, if two OSGi services are configured to use the same replication strategy, they can be installed in the same group of replicas. This solution has the advantage of reducing the number of control messages exchanged by the group communication system (for instance, two replicated services may use the same failure detector module). The selection of the group communication channel is performed at configuration time of a service. It is possible to define group communication configurations such as: group name, primary views, total order

| Configuration | Replication | Reply | State | Broadcast | Views |
|---|---|---|---|---|---|
| A | Active | First | Stateless | Total regular | Partitionable |
| B | Lazy-passive | - | Stateful | Reliable regular | Primary |
| C | Active | Majority | Stateful | Total uniform | Primary |
| D | Eager-passive | - | Stateful | Reliable uniform | Primary |

Table 3.1: Examples of configuration options for the proposed architecture.

and uniformity.

When a service is replicated using the active replication, multiple replies to the same request may be generated. In non-faulty runs, each replica will reply to the client application. FT-OSGi allows to configure how these replies are filtered before being returned to the client application. For this purpose, there is a proxy installed in the client that collects the replies from servers and returns only one answer to the application, filtering duplicate replies and simulating the operation of a non-replicated service. The FT-OSGi proxy supports three distinct modes for filtering the replies from servers: `wait-first`, `wait-all` and `wait-majority`. In the `wait-first` mode, the first received reply is received and returned immediately to the client; all the following replies are discarded. In the `wait-all` mode, the proxy waits for all the replies from the servers, compares them, and returns to the client one reply, if all the replies are equal. If there is an inconsistency in the replies, the proxy raises an exception. Finally, the `wait-majority` returns to the client one reply, as soon as a majority of similar replies are received. This filtering configurations are defined by the client, allowing each client to obtain a reliability level matching its own needs.

Distribution and replication are hidden from the clients, that always interact with a local instance of the OSGi Framework. Thanks to this approach, the semantic of the OSGi events is maintained. All the events generated by the services and the OSGi Framework itself are propagated to the clients. To simulate a non replicated service, the proxy installed in the client filters duplicated events, using the approach previously described to process the replies from the servers.

Table 3.1 shows some examples of how to configure FT-OSGi applications. It is possible to configure the replication strategy, the filtering mode of server replies, and the operation of the group communication service.

## 3.2 Building Blocks

The FT-OSGi is composed of several building blocks to support the communication between the nodes of the system and to support the consistency between replicas. The building blocks used to support communication and consistency are the R-OSGi and a Group Communication Service (GCS), that are described in the next sections.

### 3.2.1 R-OSGi

R-OSGi (Rellermeyer, Alonso, & Roscoe 2007) is a platform capable of distributing an OSGi application through several nodes in a network. R-OSGi is layered on top of the OSGi platform in a transparent way to applications, being possible to run any OSGi application in the R-OSGi platform. The R-OSGi is implemented as an OSGi bundle and service, and should be running in all available nodes that want to make available their OSGi services remotely.

The OSGi Service Platform only allows the interaction of OSGi services running in a single, local OSGi instance. With R-OSGi, the concept of OSGi remote services is introduced, which allows the interaction among OSGi services running in different OSGi instances. To achieve this goal, R-OSGi uses proxies to represent a service that is running in a remote node. A proxy consists in a normal OSGi service, which is installed in the client node when a remote service is discovered and requested by the application. The proxy is constructed on the fly, and simulates the remote service locally by implementing all the methods as remote calls to the original service. When the application invokes a method in the proxy, that proxy will issue a remote method invocation to the remote node holding the service implementation in a transparent way to application.

When two different R-OSGi nodes connect, symmetric leases are exchanged between them. A lease received from a remote R-OSGi node contains information about its services. After the first lease exchange, each node, is responsible to maintain the leases it exports up to date. Therefore, the nodes providing remote services need to send lease updates, when the exported services change. Notification about additions, modifications, or removals of services are delivered to the nodes using those services.

In addition to remote invocation, R-OSGi also supports the exchange of OSGi events among different nodes. This feature is useful for applications that can take advantage of a publisher/-

subscriber semantics. R-OSGi exchanges information about the subscribers of a specific event when a pair of leases is established between two R-OSGi nodes, and maintains this information always synchronized between them through lease updates. Using this approach, a R-OSGi node knows where the listeners of a specific event are located. Therefore, when an event occurs in its OSGi instance, R-OSGi propagates the event to the nodes that are subscribers of that event.

Since the OSGi service registry is local to each OSGi instance, it is necessary to distribute it to allow the discovery of services in other OSGi instance nodes. The approach followed by the R-OSGi uses the *Service Location Protocol* (SLP) (Guttman 1999) to provide a distributed service registry. The SLP protocol requires an URI (Berners-Lee, Fielding, & Masinter 2005) containing the location of the announced services in the network. Therefore, the R-OSGi nodes and services are identified in the network by an unique URI, using this scheme: `r-osgi://<Address>:<Port>#<ServiceID>`. This URI allows a direct and explicit connection to a remote service. The constructed proxy also uses internally this URI information to store the location of the remote service, in order to perform the remote invocations.

The communications between R-OSGi nodes are based on message exchange, and can use multiple network channels implementations. R-OSGi, by default, uses a network channel implementation that relies on a persistent TCP connection. This decoupling of the network channel permit to use R-OSGi, for example, through bluetooth communication. FT-OSGi is integrated with R-OSGi at this level.

### 3.2.2 Group Communication Service

A Group Communication Service (GCS) provides two complementary services: (*i*) a membership service, that provides information about the nodes that are in the group and generates view changes whenever a member joins, leaves or is detected as failed, and (*ii*) a group communication channel between the nodes that belong to the group membership. The FT-OSGi uses a generic service (jGCS) (Carvalho, Pereira, & Rodrigues 2006) that can be configured to use several group communication toolkits, such as Appia (Miranda, Pinto, & Rodrigues 2001) or Spread (Amir, Danilov, & Stanton 2000). The prototype presented in the dissertation uses Appia. Appia is a protocol composition framework to support communication, implemented in the Java language. The main goal of Appia is to provide high flexibility when composing communication protocols in a stack, and to build protocols in a generic way for reusing them

in different stacks. Appia contains a set of protocols that implement view synchrony, total order, primary views, and the possibility to create open and closed groups. An open group is a group of nodes that can send and receive messages from nodes that do not belong to the group. This particular feature is very important to FT-OSGi. It is also important to our system that the GCS used gives the possibility to chose the message ordering guarantees (regular FIFO for passive replication or total order for active replication), the reliability properties (regular or uniform broadcast) and the possibility to operate in a partitionable or non-partitionable group. Appia has also a service that maintains information about members of a group using a best effort approach. This service, called *gossip*, allows the discovery of group members (addresses) by nodes that do not belong to the group.

## 3.3 System Architecture and Components

Figure 3.1 depicts the FT-OSGi architecture, representing the client and server components. Each node has an instance of the OSGi platform, the R-OSGi extension for distribution, and the FT-OSGi component. The main difference between a client and a server is the type of services installed in the local OSGi platform. The servers maintain the replicated services that will be used by clients. The clients contain proxies that represent locally the services that are installed in the servers. When a client needs to access a service that is installed remotely (in a replicated server), a proxy is created to simulate the local presence of that service. In detail, each method of this proxy service invoked locally uses R-OSGi to execute a remote call to the replicated servers which have the implementation of the OSGi service.

On top of the previously described services, the components described in Sections 3.3.1 and 3.3.2 were built to provide fault tolerance to OSGi services.

### 3.3.1 FT Service Handler

At each node, this component provides the information about the services available on remote nodes that can be accessed by the local node. In particular, it provides (for each service) the FT-OSGi configuration options, such as, for instance, the replication strategy used and how replies are handled by the proxy.

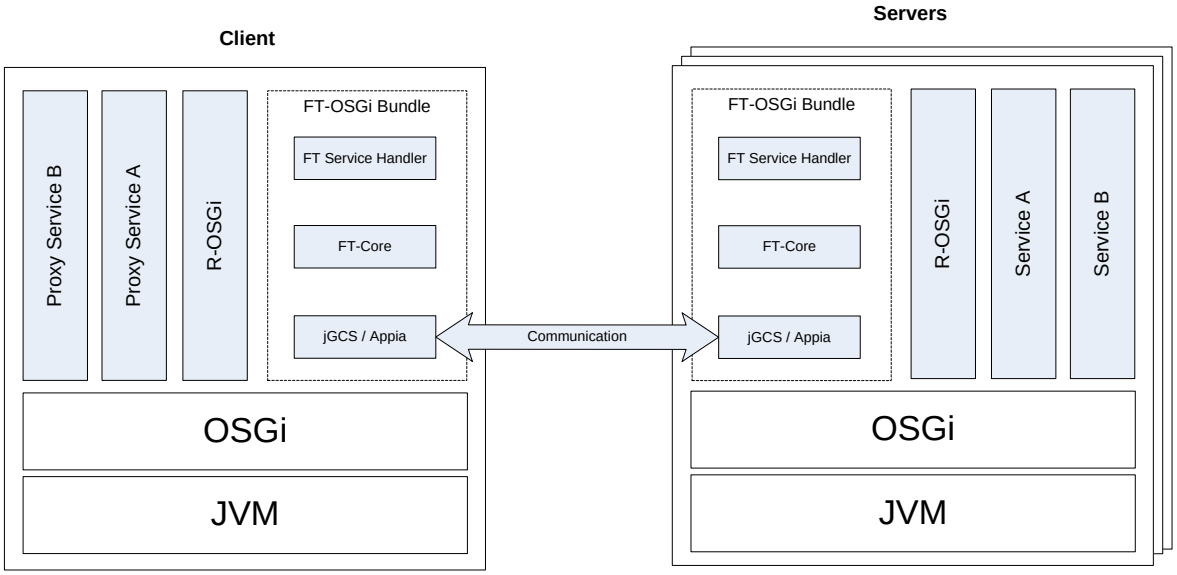



Figure 3.1: Architecture of a server and a client.

### 3.3.2 FT-Core

This component is responsible for maintaining the consistency among all the replicas of the service. It also hides all the complexity of replication from the client applications. The FT-Core component is composed by four sub-components that are described in the next paragraphs: the *Appia Channel Factory*, the *Appia Channel (Client)*, the *Appia Channel (Server)* and the *Replication Mechanisms*.

The *Appia Channel Factory* component is responsible for the definition of the replication service for an OSGi service. Each OSGi service is associated with a group of replicas, which is internally identified by an address in the form `ftosgi://<GroupName>` (this address is not visible for the clients). The group of replicas supports the communication between the replicas of the OSGi service and communication between the client and the group of replicas. The client is not a member of the group and uses the open group functionality supported by Appia. The communication between replicas is done using view synchronous communication (with total order in the case of active replication). For each replica group, an Appia channel is created. The channel to communicate among replicas is created when a service is registered with fault tolerance properties and is implemented by the *Appia Channel (Server)* component. The communication channel between clients and service replicas is created when some client needs to access a replicated service and is implemented by the *Appia Channel (Client)* component.

Finally, the *Replication Mechanisms* component implements the replication protocols used in FT-OSGi (active, eager-passive and lazy-passive replication) and is responsible for managing the consistency of the replicated services. This component is also responsible for the recovery of failed replicas and for the state transfer to new replicas that dynamically join the group. For managing recovery and state transfer, these components uses the membership service already provided by the GCS.

## 3.4 Replication Strategies

The FT-OSGi extensions support three different types of replication: *active replication*, *eager-passive replication* and *lazy-passive replication*. These three strategies are described in the next paragraphs.

### 3.4.1 Active Replication

This replication strategy follows the approach of standard active replication (Guerraoui & Schiper 1997), where each and every service replica processes invocations from the clients. When some replica receives a new request, it atomic broadcasts the request to all replicas. All the replicas execute the same requests in the same global order. A limitation of this replication strategy is that it can only be applied to deterministic services.

The interaction among the system components essentially depends on the replication strategy used by the replicated service. Figure 3.2 depicts the interaction among the several components of the FT-OSGi extensions using the active replication strategy. As an example, the client invokes a method provided by Service A, which is replicated in GroupA, and it already has an instance for the proxy that represents locally the Service A. The client starts by invoking that method on the local proxy (step 1). The service is actually deployed remotely, so the proxy invokes that call on R-OSGi (step 2). The original communication channel of R-OSGi was re-implemented to use an Appia channel, instead of a TCP connection. So, R-OSGi is actually using FT-OSGi to send the requests, through the *Appia Channel (Client)* component (steps 3 and 6). If the client does not have in its cache at least one address of the members of GroupA, it queries the *gossip* service (steps 4 and 5). This request to the *gossip* service is done periodically, in order to maintain the cache updated, and is resent until it is successfully received by one

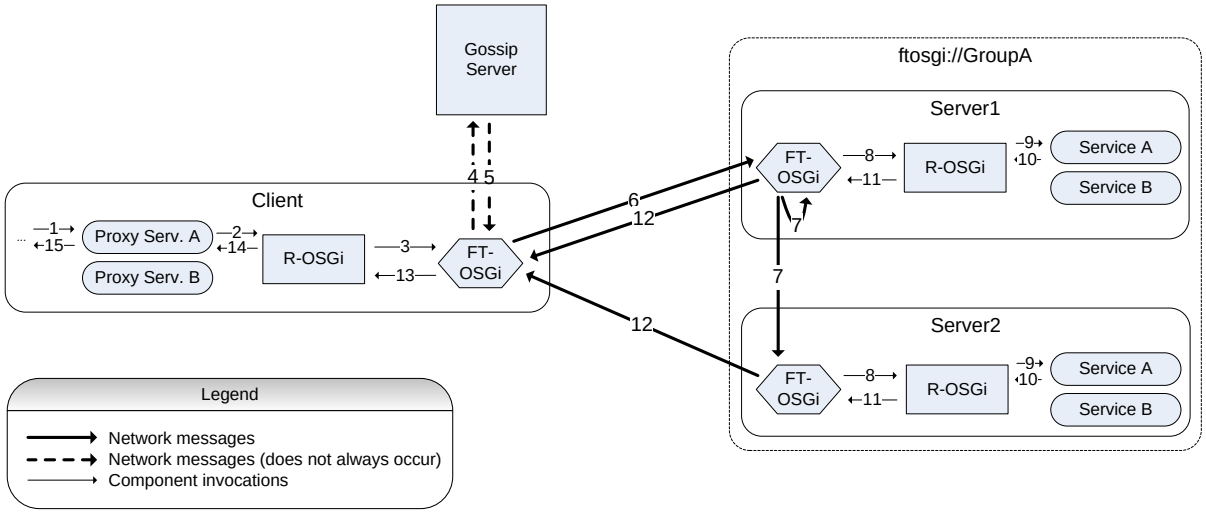



Figure 3.2: Interaction between components when using active replication.

of the servers. When one of the servers receives the client request, it atomically broadcasts the request to all the servers on GroupA, using the *Appia Channel (Server)* component (step 7). This ensures that all servers execute the requests in the same global total order. For each request that is delivered to each replica by the atomic broadcast primitive, the replica delivers that request to the local instance of R-OSGi, that will call the method on the Service A and obtain a response (steps 8 to 11). Notice that these four steps are made on all the replicas. All the replicas reply to the client (step 12) that filters the duplicate replies and returns one reply to R-OSGi (step 13). Finally, R-OSGi replies to the proxy, that will return to the client application the invocation result (steps 14 and 15).

### 3.4.2 Eager-Passive Replication

In this case only one replica, the primary, deals with invocations from the clients (Guerraoui & Schiper 1997). The primary replica is the same for all services belonging to a replica group. The backup replicas receive state updates from the primary replica for each invocation of stateful services. The primary replica only replies to the client after broadcasting the state updates to the other replicas. Attached to the state update message, the backup also receives the response that will be sent by the primary replica to the client. This allows the backup to replace the primary and resend the reply to the client, if needed.

Figure 3.3 depicts the interaction among the components of the FT-OSGi extensions us-

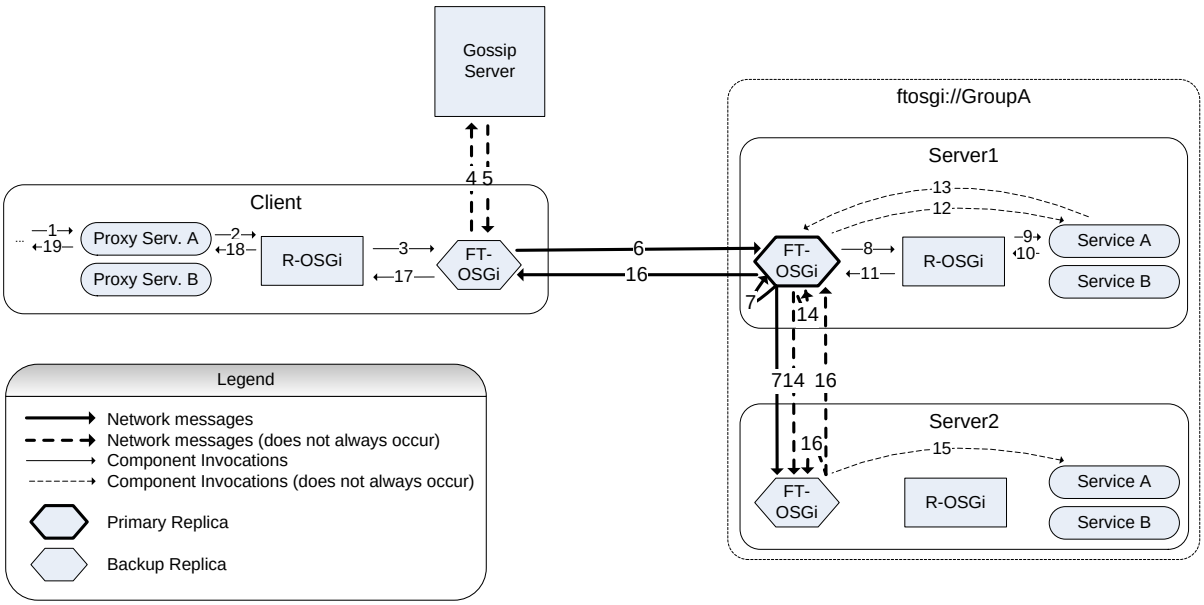



Figure 3.3: Interaction between components when using eager-passive replication.

ing the eager-passive replication strategy. We use an example similar to the example used to illustrate active replication. Steps 1 to 7 are the same as in the previous example. The first difference is that only the primary replica delivers the request to the local instance of R-OSGi, that will call the method on the Service A and obtain a response (steps 8 to 11). If Service A is declared as stateful, its state is obtained (steps 12 and 13), broadcast to all group replicas (step 14) and, at last, the Service A state on each backup replica is updated with the one received from the primary replica (step 15). As soon as the primary replica receives an acknowledgment from each backup replica of the state update (step 16), the primary replica replies to the client (step 17), which, in turn, returns the reply to R-OSGi (step 18). Finally, R-OSGi replies to the proxy, that will return to the client application the invocation result (steps 19 and 20).

### 3.4.3 Lazy-Passive Replication

This replication strategy follows the same principles of *eager-passive replication*. However, the reply to the client is sent immediately, as soon as the request is processed. The state update propagation is done in background, after replying to the client. This strategy provides less fault tolerance guarantees and raises the possibility of inconsistencies between replicas in a specific failure scenario. The failure scenario consists in the primary replica failure after sending

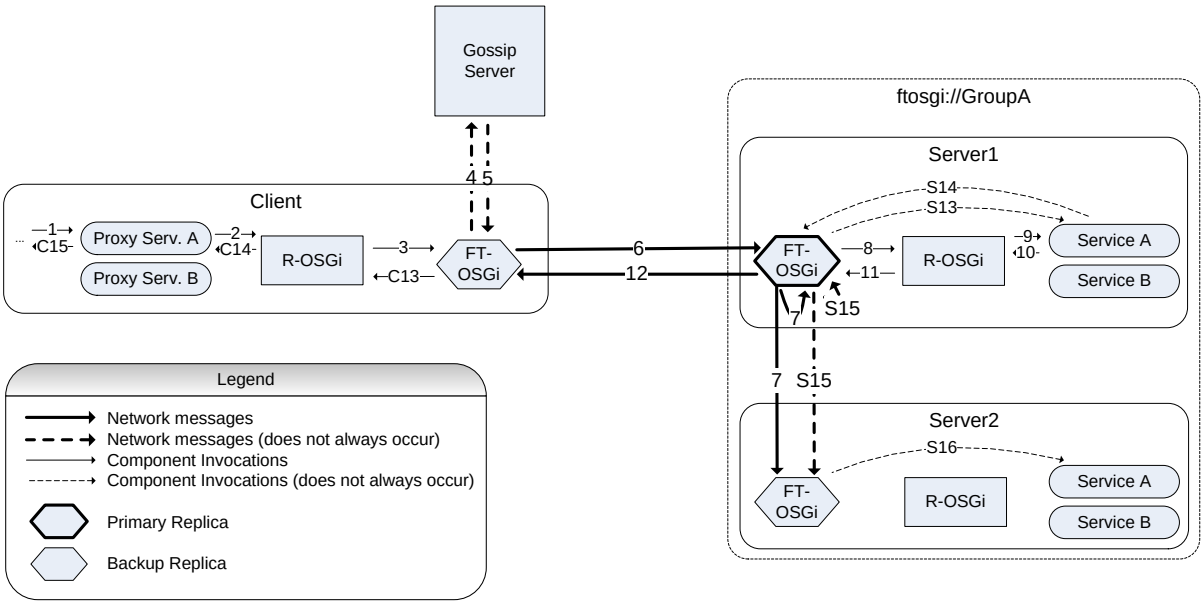



Figure 3.4: Interaction between components when using lazy-passive replication.

successfully the reply to the client and before the backup replicas receiving successfully the corresponding state update. On the other hand, this replication strategy is faster and may be adequate for many applications which not require strong guarantees.

Figure 3.4 depicts the interaction among the components of the FT-OSGi extensions using the lazy-passive replication strategy, following a similar example as in the previous replication strategies. Steps 1 to 11 are the same as the previous example of eager-passive replication. The main difference is in the timing of the response from the primary replica to the client. Right after FT-OSGi receives the response from R-OSGi (step 11), it sends the reply to the client (step 12). After this step, the following steps are performed in parallel. These steps are labeled with a prefix to the step number: in client side the prefix is `C` and in the server side is `S`. If the Service A is declared as stateful, its state is obtained (steps S13 and S14), broadcast to all group replicas (step S15) and, at last, the Service A state on each backup replica is updated with the one received from the primary replica (step S16). In the client side, after R-OSGi receives the response from the FT-OSGi (step C13), R-OSGi replies to the proxy, that returns to the client application the invocation result (steps C14 and C15).

### 3.4.4 Replica Consistency

The group of service replicas is dynamic, which means that it supports the addition and removal of servers at runtime. It also tolerates faults and later recovery of the failed replica. The following paragraphs describe the techniques used to manage dynamic replica membership.

#### 3.4.4.1 Leader Election

The replication protocols implemented in the *Replication Mechanisms* component require a mechanism for leader election for several reasons. In the case of eager-passive and lazy-passive replication, leader election is necessary to choose the primary replica, executing the requests, and disseminating the updates to the other replicas. In all replication strategies, leader election is used to choose the replica that will transfer the state to replicas that are recovering or are joining the system. The leader election mechanism can be trivially implemented on top of jGCS/Appia because upon any membership change, all processes receive an ordered set of group members. By using this feature, the leader can be deterministically attributed by choosing the group member with the lower identifier that also belonged to the previous view.

#### 3.4.4.2 Joining New Servers

When one or more replicas join an already existing group, it is necessary to update the state of the incoming replicas. The state transfer starts when there is a view change and proceeds as follows. If there are new members joining the group, all replicas stop processing requests. The replica elected as primary (or leader) sends its state to all the replicas, indicating also the addresses of the new replicas joining in the view. The state transfer also contains the services configurations for validity check purposes. When a joining replica receives, validates the services configurations and updates its own state, it broadcasts an acknowledgment to all the members of the group. Finally, when all the group members receive the acknowledgments of all the joining replicas, all resume their normal operation. During this process, three types of replica failures can occur: *i*) new joined replica failure; *ii*) primary (or leader) replica failure; *iii*) another (not new, neither primary) replica failure. To address the first type of failure, for each new joined replica that fails, the remaining replicas will discard the expected acknowledgment of that failed replica. The second type of failure only requires an action when the primary replica fails before

sending successfully the state to all new joined replicas. In this case, the new primary replica sends the state to the replicas. This solution tries to avoid sending unnecessary state messages. Regarding the third type of failure, these failures do not affect the process of joining new servers.

#### 3.4.4.3 Recovering From Faults

Through the fault detector mechanisms implemented on top of jGCS/Appia, it is possible for FT-OSGi to detect when a server replica fails. FT-OSGi treats a failure of a replica in the same way treats an intent leave of a replica from the group membership. When a replica fails or leaves, some approach is necessary to maintain the system running. If the failed or leaving replica was the leader replica (also known as primary replica for both passive replication strategies), it is necessary to run the leader election protocol to elect a new replica to play that role. Otherwise, the remain replicas just remove from the group membership the failed replica.

## 3.5 Life Cycle

This section describes how the FT-OSGi components are created on both the client and the group of servers. The FT-OSGi uses the SLP protocol (Guttman 1999) to announce the set of services available in some domain. The replication parameters are configured using Java properties. This feature allows to read the parameters, for instance, from a configuration file contained in a service. The replication parameters are: the group name that contain the replicated service, the type of replication, the group communication configuration, among others.

When a new replica starts with a new service, it reads the configuration parameters for replication and creates an instance of *Appia Channel (Server)* with the specified group name. It creates also an instance of the *FT-Core* and *Replication Mechanisms* components with the specified replication strategy. Finally, the replica registers the service in SLP, specifying the service interface and that it can be accessed using the address `ftosgi://<GroupName>`. New replicas will also create an instance of *Appia Channel (Server)*, *FT-Core* and *Replication Mechanisms*, but they will join the already existing group.

The client starts by executing a query to SLP, asking for a reference that implements a service with a specified interface. If the service exists in the system, the SLP returns the address

Listing 3.1: Server example code.

```
1  public class HelloServiceImpl implements HelloService {
2   public String speak() {
3    return "Hello World!";
4   }
5   public String yell() {
6    return ("Hello World!".toUpperCase().concat("!!!"));
7   }
8  }
9
10 public class Activator implements BundleActivator {
11  private HelloService service;
12  public void start(BundleContext context) throws Exception {
13   service = new HelloServiceImpl();
14
15   Dictionary<Object, Object> properties = new Hashtable<Object, Object>();
16   properties.put(RemoteOSGiService.R_OSGi_REGISTRATION, Boolean.TRUE);
17   properties.put(FTServiceTypes.FT_ROSGi_REGISTRATION, Boolean.TRUE);
18   properties.put(FTServiceTypes.FT_ROSGi_FT_SERVICE_ID, "HelloService");
19   properties.put(FTServiceTypes.FT_ROSGi_FT_TYPE, FTTypes.ACTIVE_STATELESS);
20   properties.put(FTServiceTypes.FT_ROSGi_FT_GROUPNAME, "GroupA");
21   properties.put(FTServiceTypes.FT_ROSGi_PRIMARY_VIEW, Boolean.TRUE);
22
23   context.registerService(HelloService.class.getName(), service, properties);
24  }
25  public void stop(BundleContext context) throws Exception {
26   service = null;
27  }
28 }
```

where the service can be found, in the form `ftosgi://<GroupName>`. In a transparent way to the client application, FT-OSGi creates a proxy that will represent locally the service and an instance of *Appia Channel (Client)* to send messages (requests) to the group of replicas of the service. After creating these components, a reference of the proxy is returned to the client application.

## 3.6 Programing Example

This section illustrates how a client application and a service can be implemented using the FT-OSGi extensions.

We will start by showing how to implement and configure a service. Listing 3.1 shows a typical *HelloWorld* example implemented as an OSGi service. The service implements an interface (`HelloService`) with two methods that are deterministic (lines 1 to 8). After implementing the service, it must be configured and registered in the OSGi platform. This is done

Listing 3.2: Client example code.

```
1  public class Activator implements BundleActivator {
2
3   public void start(BundleContext context) throws Exception {
4    final ServiceReference ftRef = context.getServiceReference(FTFramework.class.getName());
5    if (ftRef == null) {
6      System.out.println("No FTFramework found!");
7      return;
8    }
9    FTFramework ftFramework =(FTFramework)context.getService(ftRef);
10
11   URI helloURI = ftFramework.getFTServiceURI(HelloService.class.getName());
12   HelloService helloService =
13    (HelloService)ftFramework.getFTService(HelloService.class.getName(), helloURI);
14   if(helloService == null) {
15    System.out.println("No HelloService found!");
16    return;
17   } else {
18    // Can start use the service requested
19    System.out.println("Response: " + helloService.speak());
20   }
21  }
22
23  public void stop(BundleContext context) throws Exception {}
24 }
```

in the class `Activator`, where it can be seen that the service is configured to use active replication, it is stateless, uses primary views, and it belongs to the group with the following address `ftosgi://GroupA` (lines 15 to 21). The registration of the service in the OSGi platform makes the service available to its clients and is done after the configuration process (line 23).

Listing 3.2 shows an example of how a client application can obtain the instance of the replicated `HelloService` previously registered by the servers. First of all, in the class `Activator`, the application starts to obtain the local service `FTFramework` (lines 4 to 9). This `FTFramework` service is responsible to abstract the interaction with the SLP. Using that service, the application obtains the address `ftosgi://GroupA` (line 11), corresponding to the address where is located the `HelloService`. Notice that this address is actually in an opaque object of type `URI`, so it could be also an address of a non-replicated service (for instance, a R-OSGi service). Afterwards, with that address, the application can request the service instance (lines 12 and 13), which if it is successfully executed will create and register a service proxy of the `HelloService` in this local OSGi instance. Then, the proxy instance is returned to the client application, and that instance can be used like any other OSGi service. In this example an invocation of the method `speak()` is executed (line 19), which follows the invocation procedure of an actively replicated

service, as described in the Section 3.4.

## 3.7 OSGi Implementation Details

This section focus on presenting some relevant implementation details of the OSGi Service Platform, which are required to later understand our solution. It starts detailing the components and properties of the OSGi Services. Subsequently, some details and functionalities of the OSGi Events are described.

### 3.7.1 OSGi Services

The OSGi services are defined by two components: a service interface, and a service object. The service interface corresponds to a public Java interface class, and should reflect as little implementation details as possible. The service object corresponds to the implementation of a service interface, and is owned by a specific OSGi bundle. The service is automatically unregistered by the OSGi Framework when the bundle owner is stopped or uninstalled. Therefore, the OSGi Framework must maintain information regarding the bundle owner of the service object and regarding the bundles using it.

The OSGi services may include several properties which provide useful information about the service object (we recall that, in the OSGi context, the services properties are structures holding configurations). The OSGi Framework, by default, adds some properties to every service, but it is possible to add several others. By default the OSGi Framework adds the following properties to all registered services: an object class and a service id (SID). The object class property is set by the OSGi Framework with the interface names of the service object. The SID property contains an unique service id for each registered service object, which is assigned by the OSGi Framework.

These implementation details are relevant to understand our solution, because an issue emerges from this OSGi mechanisms. The SID property cannot be a service unique identifier in a replicated system.

### 3.7.2 OSGi Events

The OSGi Framework provides the possibility to developers to use events. To support this feature, the OSGi specification defines the Event Admin Service (OSGi Alliance 2007b). The events provide to the developers the usage possibility of the publish/subscribe model, which is useful in many applications. To watch publishers and subscribers, the OSGi Service Platform uses the whiteboard pattern (Kriens & Hargrave 2004). This strategy consists on using the OSGi service registry to know the subscribers of a specific event, instead of overloading the publishers with that role. In this way, the process of sending and receiving events is simplified.

#### 3.7.2.1 Events, Handlers, and Publishers

Each OSGi event is defined by two attributes: topic and properties. The topic defines the type of the event, which allows the subscribers the possibility to choose the types of events they are interested in receiving. Therefore, the topic is the first-level filter used to verify which subscribers should receive the event. The topic is defined with a `String` and should be a hierarchical name space, e. g. `org/osgi/framework/FrameworkEvent/INFO`. The properties provide all the detailed information about the event. Therefore, the core information of the event is in its properties.

The subscribers of events, known as event handlers, receive events sent by event publishers. An event handler consists in a service that implements the interface `org.osgi.service.event.EventHandler` and must define the topics which is interested in. The event handler can define multiple topics and can even use a wildcard `*` in the last token of a topic name to allow receiving all events matching the tokens before, e. g., `org/osgi/*` matches events with topics, such as: `org/osgi/framework/FrameworkEvent/INFO`, `org/osgi/framework/FrameworkEvent/ERROR`, among others. The event handler can also define filters based on the properties of the event.

The event publishers, in order to send events, should retrieve the Event Admin Service. Then, the Event Admin Service is responsible to deliver the events to the event handlers that match the event topic, obtaining that information from the OSGi service registry.

#### 3.7.2.2 OSGi Events

The OSGi Service Platform by itself uses events for some of its functionalities. Those events can be divided in three categories: *Framework Event*, *Bundle Event* and *Service Event*. Each category has specific types of events, e. g., in the *Service Event* category exists three types of events:

- `org/osgi/framework/ServiceEvent/REGISTERED`: occurs when a new service is registered in OSGi;
- `org/osgi/framework/ServiceEvent/MODIFIED`: occurs when an existing service in OSGi is modified;
- `org/osgi/framework/ServiceEvent/UNREGISTERED`: occurs when an existing service is unregistered from OSGi.

A more detailed description of the OSGi events is described in OSGi Service Platform Service Compendium (OSGi Alliance 2007b).

## 3.8 FT-OSGi Implementation Details

This section focus on the most important implementation details, issues, and challenges that appeared during the development and implementation of FT-OSGi. Then, it describes some performance improvements made to the initial FT-OSGi implementation. Afterwards, it presents details about the messages exchanged in the network and, finally, the log garbage collector mechanisms.

### 3.8.1 OSGi Service ID Issue

Each OSGi service registered in an OSGi instance has a service id (`SID`) assigned by the OSGi Framework itself. This SID is an `Long` object and identifies the service in a unique manner. R-OSGi uses this SID to identify remote services in different nodes. When applying this concept to a service replicated through several nodes, an issue emerges. The SID no longer identifies uniquely each service in all nodes, because the SID can be assigned differently in each node by

the local OSGi instance. To address this issue, FT-OSGi defines a replicated service id (`RSID`), which is defined by the service developer in the same way as the other service configurations, through service properties. The RSID is a `String` object, that allows the developer to define its own name space. The integration of RSID with R-OSGi is transparent, FT-OSGi always converts each RSID to the local OSGi SID in each replicated service interaction.

### 3.8.2 Filtering Replicated Events

The OSGi event mechanisms support the use of the publisher/subscriber paradigm. When replicating services, different replicas will generate multiple copies of the same event. These copies need to be filtered, to offer the client the same semantics of a non-replicated service. FT-OSGi addresses this issue using a similar approach as for filtering repeated replies in active replication, i. e., the FT-OSGi component in the client is responsible to filter repeated events from the servers. The difficulty here is related with the possibility of non-deterministic events generation by different replicas. In a non replicated system with R-OSGi, an unique id is associated with a event. In a replicated system is difficult to ensure the required coordination to have different replicas assign the same identifier to the replicated event. Therefore, the approach followed in FT-OSGi consists in explicitly comparing the contents of every event, ignoring the local unique id assigned independently by each replica. This approach avoids the costs associated with the synchronization of replicas for generating a common id for each event.

### 3.8.3 Performance Improvements

Initially, in the early tests of the prototype, the overall performance was somehow poor in comparison with the system without replication. This aspect requested a diagnose to identify where was the bottleneck in the prototype. A profiler evaluation of the prototype showed that the serialization of the objects to messages was very time consuming. The main cause for this problem was the automatic `Serialization` mechanism of Java, which generates a great deal of unnecessary information (for this specific system) to been serialized and consequently sent through the network. Therefore, reducing the amount of information to be serialized provides two main improvements to the system performance: the serialization is faster, and the time spent sending the serialized message is also improved. These optimizations required to change the use of the `Serializable` interface in the messages to the use of the `Externalizable`. The

`Externalizable` interface allows a better control of the information to be serialized for an object. Therefore, it was necessary to implement two methods in each message object: one to serialize the object, and another to read and construct the object. With this approach was possible to improve the prototype overall performance by 30%.

### 3.8.4 Network Messages

The types of messages exchanged between nodes in the network are defined as presented in Tables 3.2, 3.3, 3.4 and 3.5. The first line corresponds to the field name, and the second line the size, in bytes, of that field. Fields with variable size are represented by the ‘‘-’’ symbol. The fields represented in italic, in some situations, are not included in the message.

| Client IP | Client Port | Client Message Size | Client Message |
|---|---|---|---|
| 4 bytes | 4 bytes | 4 bytes | - |

Table 3.2: Client Message to Group (CMG).

Table 3.2 shows the contents present in a `Client Message to Group (CMG)` message. This message type is sent by the server who receives a message from the client, broadcasting it to the group of replicas. In Figure 3.2 this message type is sent in step 7. The fields contained in this message type are the following: Client IP, Client Port, Client Message Size and Client Message. The `Client IP` and `Client Port` correspond to the address of the client who is sending the `Client Message` to the server replicas. The information about the address of the client is necessary to attach to the original client message, because the server replicas need that information to know where to respond after processing the client message.

| Number of Replicas | Message Size | Message |
|---|---|---|
| 4 bytes | 4 bytes | - |

Table 3.3: Server Message to Client (SMC).

Table 3.3 shows the contents present in a `Server Message to Client (SMC)` message. This message type is sent by the server replicas when are replying to the client. In Figure 3.2 this message type is sent in step 12. The fields contained in this message type are the following: Number of Replicas, Message Size and Message. The `Number of Replicas` corresponds to the

actual number of replicas joined to the communication group of the server replica replying to the client. This information is crucial for the client, in order to know how many responses has to wait in the `wait-majority` and `wait-all` filtering modes. The `Message` is the reply of a server replica to the client.

| StateID | RSID Size | RSID | IP | Port | RespFlag | *Response* | State |
|---|---|---|---|---|---|---|---|
| 4 bytes | 4 bytes | - | 4 bytes | 4 bytes | 1 byte | - | - |

Table 3.4: Service State Update Message (SSUM).

Table 3.4 shows the contents present in a `Service State Update Message (SSUM)` message. This message type is sent by the primary replica to all backup replicas in the passive replications strategies. In Figure 3.3 this message type is sent in step 14. The fields contained in this message type are the following: StateID, RSID Size, RSID, IP, Port, RespFlag, *Response* and State. The `StateID` corresponds to a unique identifier of this state update message. The `RSID` (replicated service id) is the unique identifier of the service that will be updated with the new `State`. The `IP` and `Port` represents the address of the server which sent the message (primary replica), in order to reply with an acknowledgment. The `RespFlag` informs if the field `Response` is present in the message or not. The `Response` is a message containing the response of a given request that will be sent to the client by the primary replica.

| R. Size | Replicas | LSSUM Size | *List SSMU* | LCMG Size | *List CMG* | LSC Size | List SC |
|---|---|---|---|---|---|---|---|
| 4 bytes | - | 4 bytes | - | 4 bytes | - | 4 bytes | - |

Table 3.5: State Update Message (SUM).

Table 3.5 shows the contents at the `State Update Message (SUM)` message. This message is sent, by the leader replica, during the join of one or more new replicas to the communication group. The eight fields contained in this message are the following: R. Size, Replicas, LSSUM Size, *List SSMU*, LCMG Size, *List CMG*, LSC Size, and List SC. The `R. Size` corresponds to the number of new replicas, and the `Replicas` contains all the new replicas addresses joined in this interaction. This information is crucial for the new joined replicas, because they do not have the knowledge of the other new joined replicas. The `LSSMU Size` and `LCMG Size` define the number of elements in `List of SSMU` and `List of SSMU`, respectively. The `List of SSMU` contains messages `SSMU` with the last state of each stateful services that are in the communication

group. The `List of CMG` contains `CMG` messages with all the logged lease messages from the client. This ensures that leases are updated and synchronized in new replicas. The `LSC Size` contains the number of elements in `List SC`, which has all the service configurations for validity check purposes.

### 3.8.5 Garbage Collection of FT-OSGi Logs

FT-OSGi needs to maintain some temporary information to function properly. Therefore, that information must be garbage collected when no longer necessary, leading to a better performance and less memory occupation. The garbage collection mechanisms perform this task.

An issue emerge when it is not possible to determine exactly when to delete the information. The automatic timeout cleaners was the approach followed in FT-OSGi.

#### 3.8.5.1 Client Log

One of the temporary information maintained by the clients is the *MessagesReceived* structure. The *MessagesReceived* is a hash map containing the list of messages received for each message id (`XID`). This information is necessary to provide the filtering mechanisms: `wait-first`, `wait-all` and `wait-majority`. It is also necessary to ensure that none repeated messages are delivered to the application, maintaining the expected behavior of a non-replicated service.

Since the `XID` field in messages has a size limit, the `XIDs` may be reused in different future requests. Hence, if the *MessagesReceived* were never purged, after some time new messages would be rejected. Therefore, FT-OSGi defines a life timeout for each message.

#### 3.8.5.2 Server Log

One of the temporary information maintained by the servers is the *XIDsReceived* structure. The *XIDsReceived* is a FIFO list with the `XID` fields of every message received, and also information about the client address and the OSGi service requested. This information is necessary to ensure that no repeated messages are delivered to the servers, avoiding servers from processing the same request more than once. Furthermore, the remaining information is used to know the destination of the responses to that messages received.

This kind of information presents the same problem as in the previous client case. Therefore, an analogous solution, using life timeouts, was applied to purge this structure.

## Summary

This chapter presented our solution, named FT-OSGi, which is a set of extensions to the OSGi platform to improve the reliability and availability of OSGi applications. In this chapter, the building blocks and the system architecture were described. Then, the available replication strategies have been presented, namely: active, eager-passive and lazy-passive. Afterwards, we have shown how the components are created, and also a small programming example of a client side and a server side. At last, we have discussed some relevant implementation details.

# 4 Evaluation

This chapter presents an evaluation of the FT-OSGi extensions. It starts by characterizing one of the main target application areas for FT-OSGi, namely domotic applications. We then introduce a set of simple services that are used to perform the experimental evaluation, and also, we describe our testbed. The performance evaluation addresses the following metrics: service response times (remote client perspective), service state update times, non processing requests times, new replica join process times, failure detection and recover times.

## 4.1 Domotic Systems

Domotics is one of the application areas of the OSGi Service Platform. The domotic area is characterized by several different devices, with different standards, connected and interacting with each other in order to provide a better and easier control of home appliances. Examples of such devices are the following: TV's, air conditioners, Internet gateways, printers, lights, refrigerators, personal computers, etc. It is desirable to have inter-connection capabilities between all these home devices, to simplify their coordination. Unfortunately, usually, each device may use different standard of interaction, which defeats the inter-connection requirement. Examples of standards used in home devices are the following: X10, KNX, Zigbee, UPnP. As an solution to overcome this heterogeneity, several systems in the domotics area are using architectural solutions based in OSGi technology (Dobrev, Famolari, Kurzke, & Miller 2002; Li & Zhang 2004; Wu, Liao, & Fu 2007; Lin, Hsu, Chun, & Cheng 2008). The OSGi can provide powerful possibilities for home devices, allowing a better integration of several standards and the capability of providing a Service Oriented Architecture (SOA) (Bell 2008).

Figure 4.1 presents an OSGi-based architecture for domotic systems. This kind of architectures has a main problem: the OSGi gateway is a single point of failure in the system. Experiments with real everyday domotic system, identified system failures has one of the main

impairments to user satisfaction (Kaila, Mikkonen, Vainio, & Vanhala 2008). Therefore, is of paramount importance improve the domotic systems availability and reliability. Since it is expected that more and more OSGi-compliant devices will be available in the future (Wu, Liao, & Fu 2007), the usage of two or three OSGi gateways in a home can be a solution to achieve a fault tolerant domotic system.

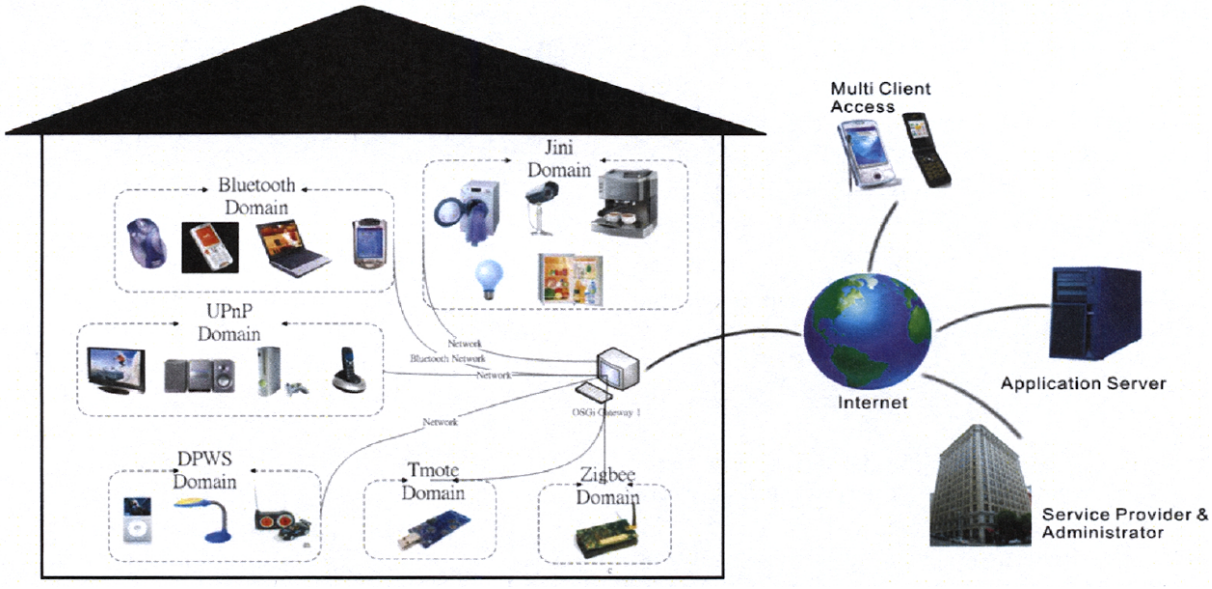



Figure 4.1: OSGi-based domotic system architecture.

Figure 4.2 depicts a possible scenario of a fault-tolerant domotic system, using two OSGi gateways with FT-OSGi extensions. This solution can be applied to the architecture of Figure 4.1 to solve the single point of failure problem of the OSGi gateway. Considering this solution and architecture of a possible domotic system, the evaluation realized to the FT-OSGi prototype had in consideration some aspects present in this type of systems. For instance, in the evaluation was considered that two or three replicated OSGi gateways would be enough and plausible in a real domotic system. It was also considered that the network will not present partitions. Another interesting aspect considered for this area is the execution time in the OSGi gateways between a request and the consequent reply. For instance, the X10 protocol, used to control lights and similar home devices, has a transmission rate around 60 baud (Shwehdi & Khan 1996). Therefore, for evaluation purposes, it is considered execution times of $2ms$ and $5ms$.

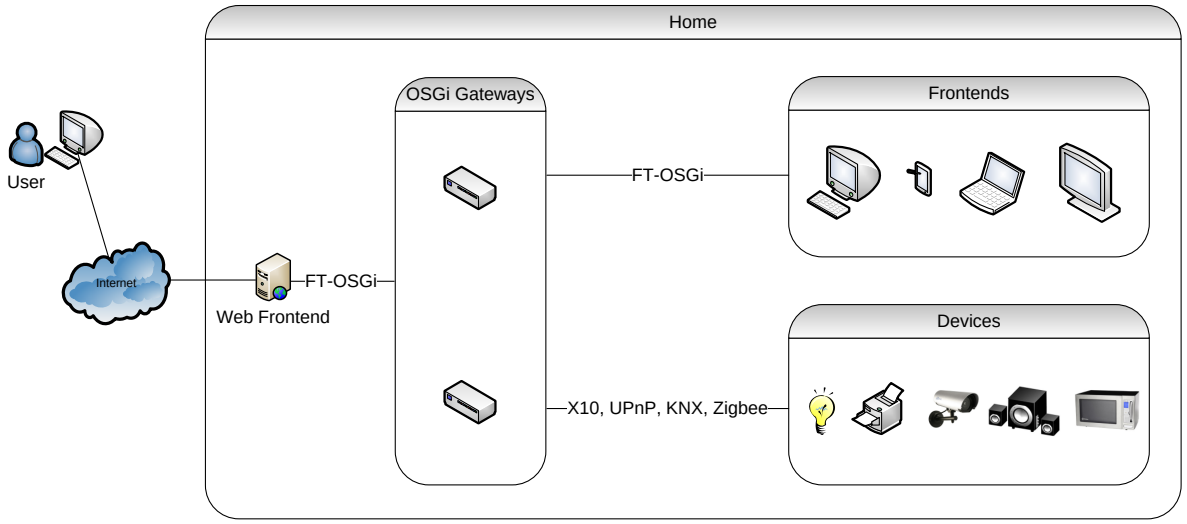



Figure 4.2: Fault-tolerant domotic system architecture.

## 4.2 Evaluation Environment

The machines used for the tests are connected by a 100Mbps Ethernet switch. The tests run in three FT-OSGi servers and one client machine. The servers have two Quad core processors Intel Xeon E5410 @ 2.33 Ghz and 4 Gbytes of RAM memory. One of the machines was also responsible for hosting the Appia *gossip* service, which is a light process that does not affect the processing time of the FT-OSGi server. The client machine has an Intel Pentium 4 @ 2.80 Ghz (with Hyperthreading) processor and 2 Gbytes of RAM memory. All the server machines are running the Ubuntu Linux 2.6.27-11-server (64-bit) operating system and the client machine is running the Ubuntu Linux 2.6.27-14-server (32-bit) operating system. The tests were made using the Java Sun 1.6.0_10 virtual machine and the OSGi Eclipse Equinox 3.4.0 platform.

All the tests of Section 4.3, Section 4.4 and Section 4.5 measure the time (in milliseconds) between the request and the reply of a method invocation on a OSGi service. In R-OSGi, the test was performed by invoking a service between a client and a server application. In FT-OSGi, different configurations were considered. The client issues a request to a group of two or three replicas. Group communication was configured using reliable broadcast in the case of passive replication and atomic broadcast (reliable broadcast with total order) in the case of active replication. Both group communication configurations used primary view membership.

## 4.3 Replication Overhead

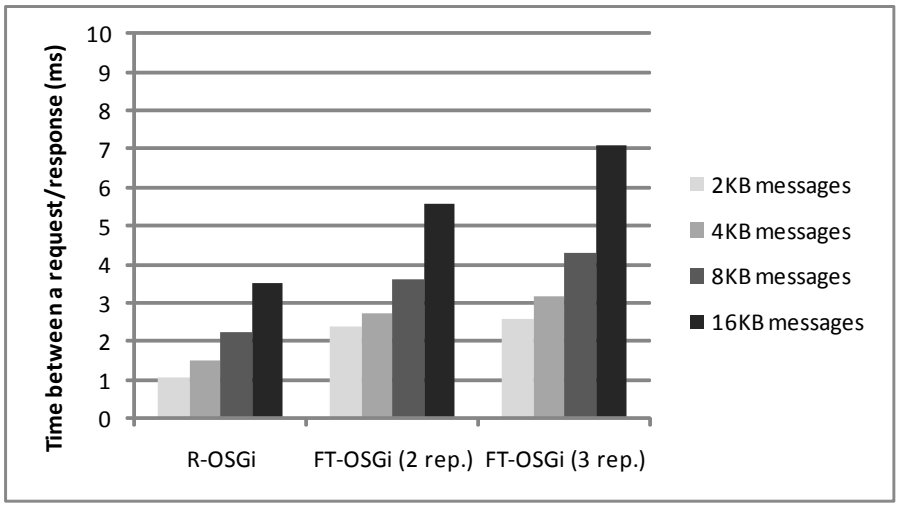
This section presents the replication overhead with different replication strategies, message sizes, execution times, and number of replicas. All tests for the active replication strategies of Figure 4.3(c), Figure 4.4(c) and Figure 4.5(c) were executed using the *wait-first* reply filtering mode. The tests for both eager-passive and lazy-passive replication strategies of the Figure 4.3(b), Figure 4.4(b), Figure 4.5(b), Figure 4.3(a), Figure 4.4(a) and Figure 4.5(a) were executed using a stateful service with 32 bytes of state. It was measured the response times with message sizes of 2 KBytes, 4 KBytes, 8 KBytes and 16 KBytes.

Figure 4.3 shows the overhead of replication on active, eager-passive and lazy-passive replications. In these tests, the execution time of the invoked method is zero. Thus, all the delays are due to remote method invocation and inter-replica coordination. The overhead of the replicated service is due to the extra communication steps introduced to coordinate the replicas. In the case of R-OSGi, where the service is located in another machine, but it is not replicated, there are two communication steps: request and reply. When using FT-OSGi, there are two extra communication steps for coordinating the replicas. In the case of the eager-passive replication (Figure 4.3(a)), there is an additional overhead due to the dissemination of the new service state to the backup replicas. As expected, the lazy-passive replication is the one with the lowest overhead (Figure 4.3(b)). It can also be observed that the message size has a similar impact on both the R-OSGi and FT-OSGi. On the other hand, as expected, adding extra replicas causes the overhead to increase.

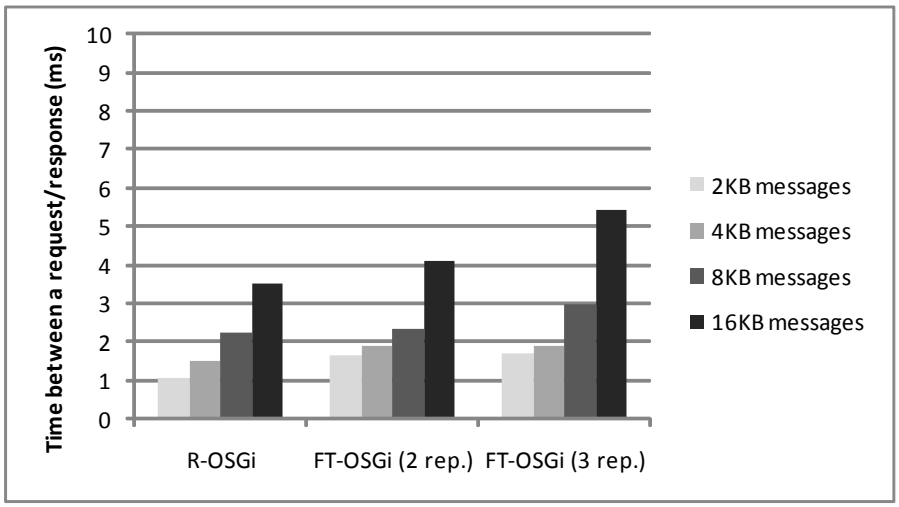
In the tests presented in Figure 4.4, the execution time of the invoked method is $2ms$. From the results it can be seen that the overhead is smaller in all replication strategies, meaning that the execution time dominate the overhead of replication. This effect is more obvious when the execution time is $5ms$, as depicted in Figure 4.5.

## 4.4 Response Filtering Modes on Active Replication

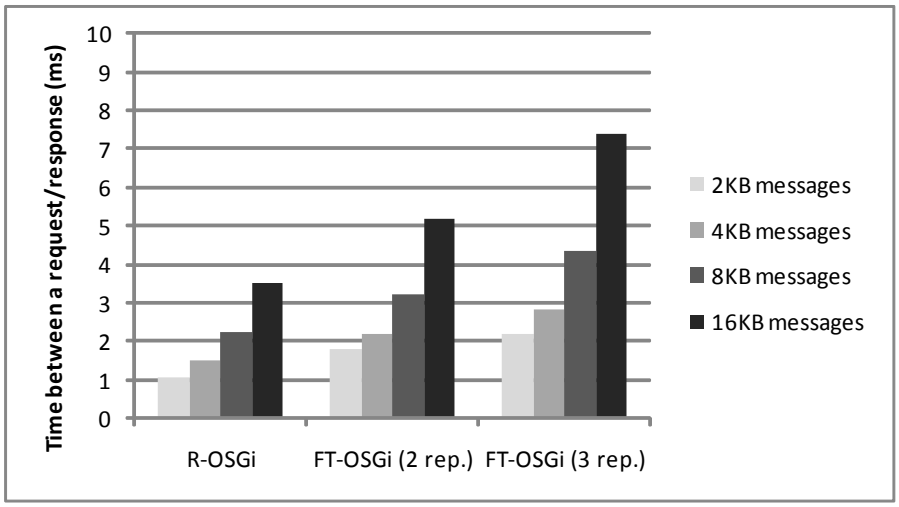
The Figure 4.6 shows the overhead in the case of active replication with the three reply filtering modes: *wait-first*, *wait-majority* and *wait-all*. The tests were configured to call a service method that takes $2ms$ to execute. The performance of a replicated system with two and three




(a) Eager-passive replication.

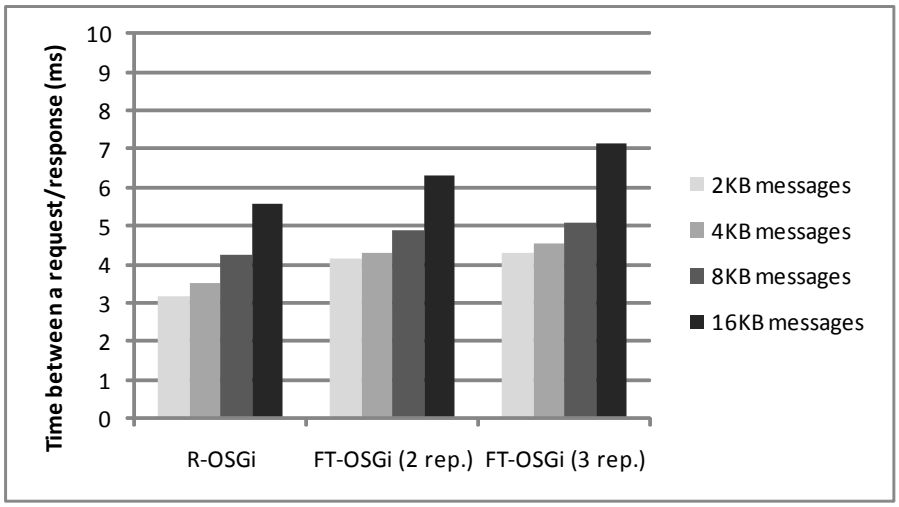



(b) Lazy-passive replication.

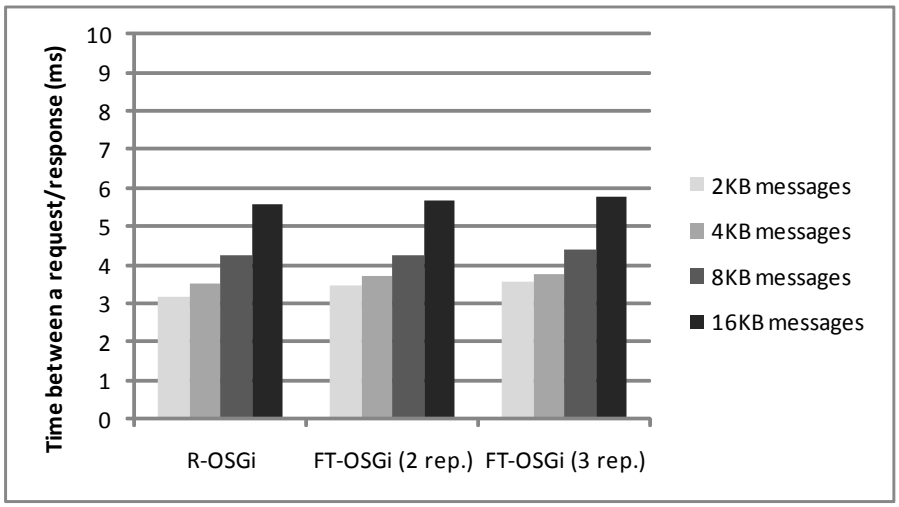



(c) Active replication.

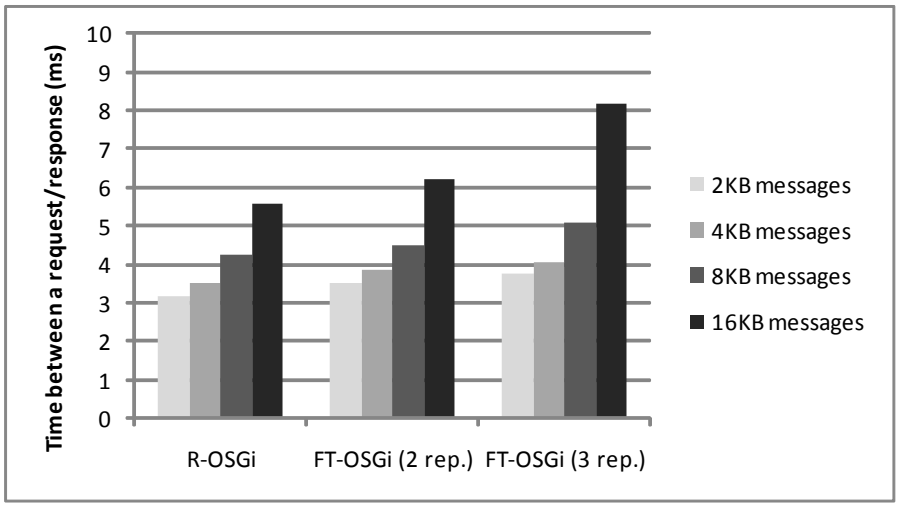
Figure 4.3: Replication overhead with no execution time.




(a) Eager-passive replication.

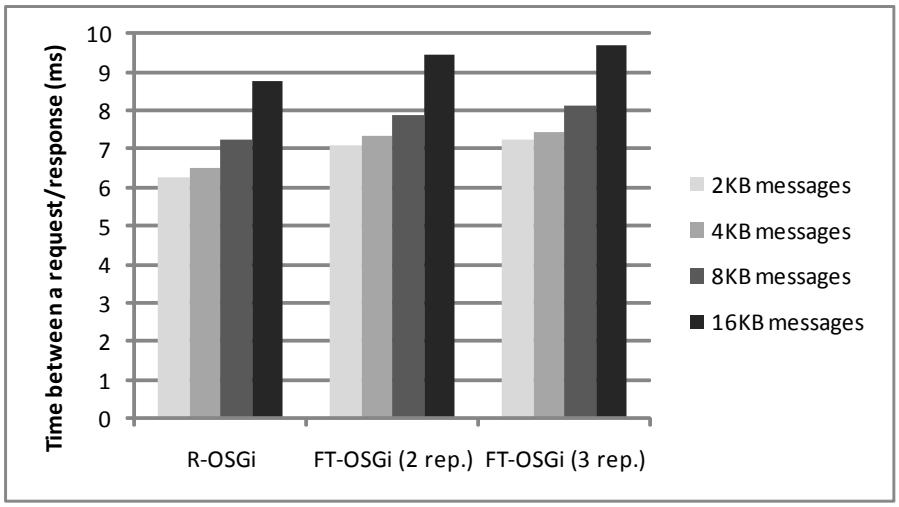



(b) Lazy-passive replication.

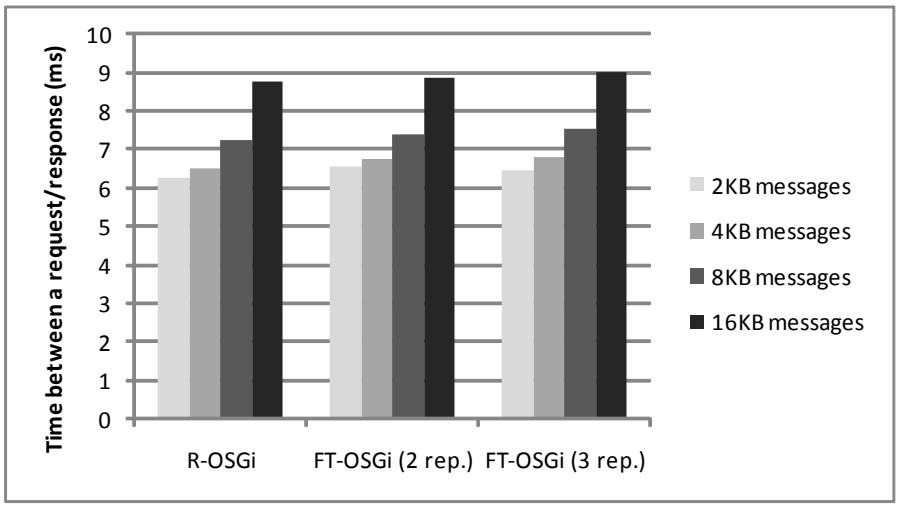



(c) Active replication.

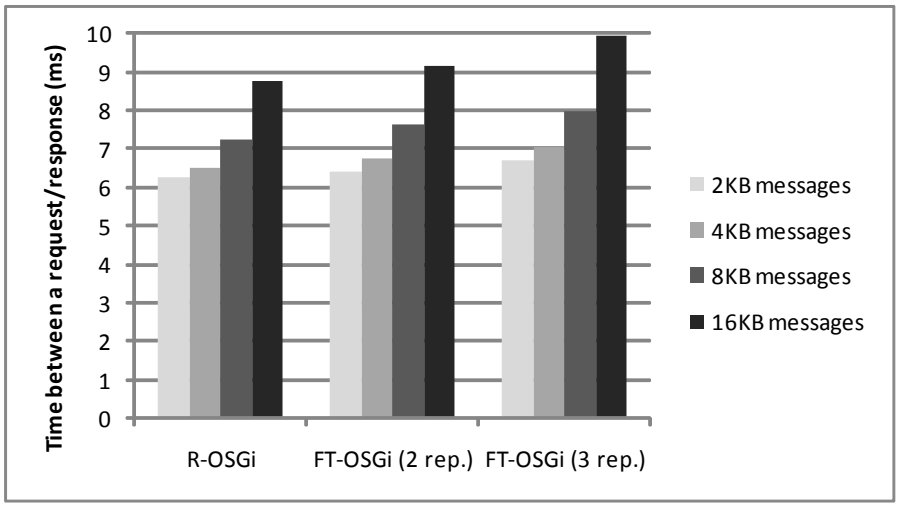
Figure 4.4: Replication overhead with an execution time of $2ms$.




(a) Eager-passive replication.




(b) Lazy-passive replication.




(c) Active replication.

Figure 4.5: Replication overhead with an execution time of $5ms$.

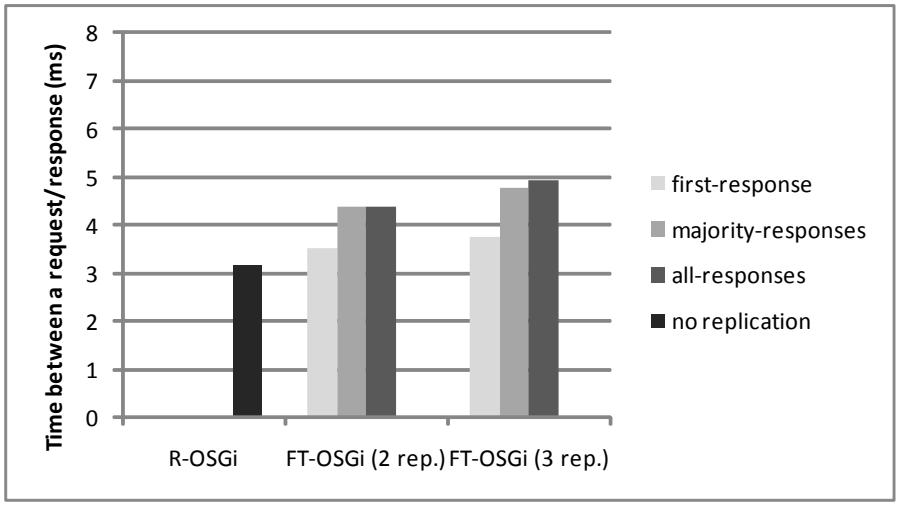
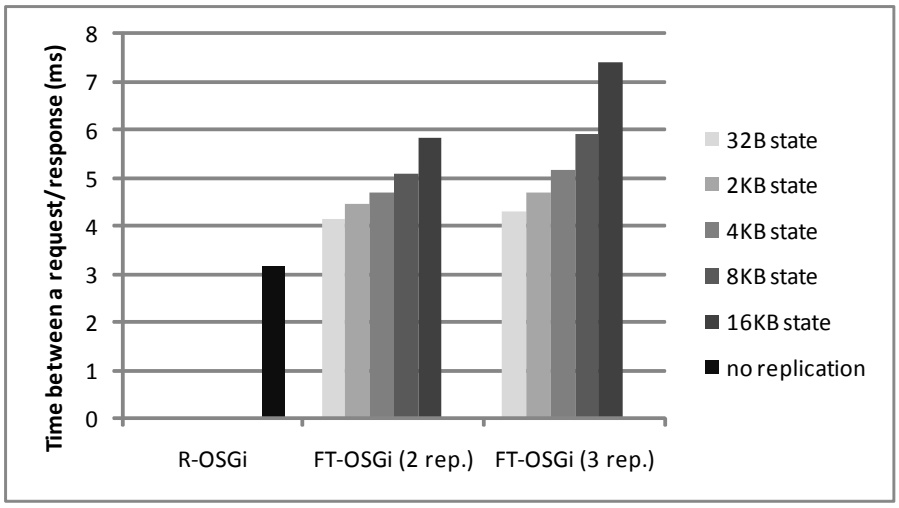



Figure 4.6: Response time on active replication with the 3 filtering modes.

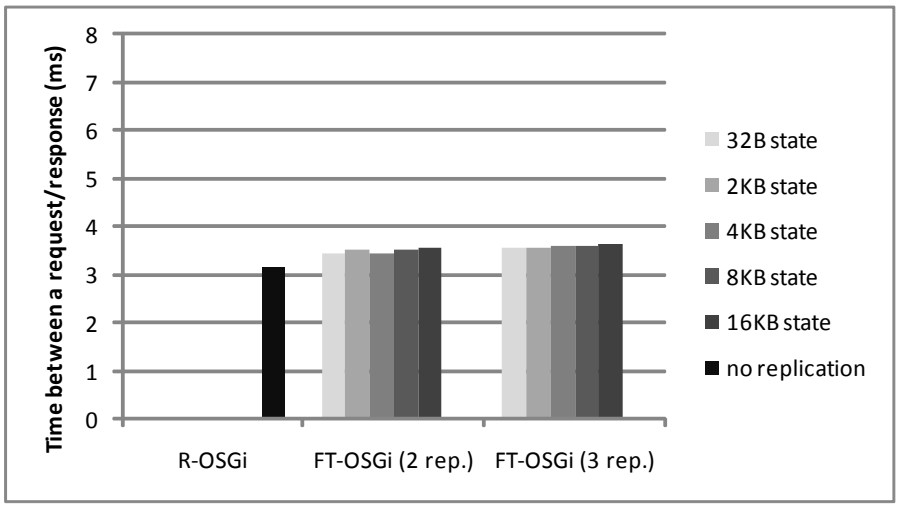
replicas was compared with R-OSGi (no replication). As it can be observed, the *wait-first* filtering mode does not introduce a large overhead when compared with R-OSGi. This can be explained by the fact that most of the requests are being received by the same node that orders the messages in the total order protocol. When the tests are configured to use the *wait-majority* and *wait-all* modes, the delay introduced by the atomic broadcast primitive is more noticeable. The results also show that the response time increases with the number of replicas.

## 4.5 Effect of State Updates on Passive Replication

Figure 4.7 presents the results when using passive replication with a stateful service. We measured the response time with a method execution time of $2ms$, and with different service state sizes of 32 bytes, 2 Kbytes, 4 Kbytes, 8 Kbytes and 16 Kbytes. The Figure 4.7(a) depicts the response times for the eager-passive replication, where the state size has a direct impact in the replication overhead. On the other hand, in the lazy-passive replication (Figure 4.7(b)), since the state transfer is made in background, in parallel with the response to the client, the state size has no direct impact on the response time.




(a) Eager-passive replication.




(b) Lazy-passive replication.

Figure 4.7: Passive replication with different state sizes.

## 4.6 Dynamic Group Membership Overhead

This section presents the overhead induced by the dynamic group membership, allowing replicas to join and leave the group at any time. To obtain the overhead of the procedure for joining a new replica we measured the duration of two different steps (in milliseconds): *i*) the time during replicas do not process client requests; *ii*) the overall time of the join process, beginning at the join request. The experiments have been executed 10 times and the results averaged. The tests were executed with: a stateless service; a stateful service varying the state size (32 bytes, 2 Kbytes, 4 Kbytes, 8 Kbytes and 16 Kbytes); and the join of a replica to an existing group with one and two replicas.

Figure 4.8 presents a graph with the average time during replicas do not process client requests, with the conditions described in the previous paragraph. Figure 4.9 presents a graph with the average duration of the overall join process, in the same conditions. Observing the results, it is clear that the state size of the services and the replica group size have a direct impact in the measured intervals. As expected, the presence of larger service state sizes results in higher join procedure times. Higher number of replicas in the group have a similar effect.

From a client perspective, the worst effect of the joining procedure, corresponds to the period during replicas do not process client requests. This period should be the lowest possible in order to maximize availability. Therefore, the addition of new replicas, preferably, should be performed when the system is lightly loaded. The worst measured time in the tests was $26 ms$.

## 4.7 Failure Handling Overhead

Failure handling in FT-OSGi is based on the underlying view-synchronous group communication mechanisms. When a replica fails, the failure is detected and the replica expelled from the replica group. FT-OSGi ensures the availability of the service as long as a quorum of replicas remains active (depending on the reply filtering strategy, a single replica may be enough to ensure availability). Failure handling is performed in two steps. The first step consists in detecting the failure, which is based on timeouts. This can be triggered by exchange of data associated with the processing of requests or by a background heartbeat mechanism. When a failure is detected, the group communication protocol performs a view change, that requires the

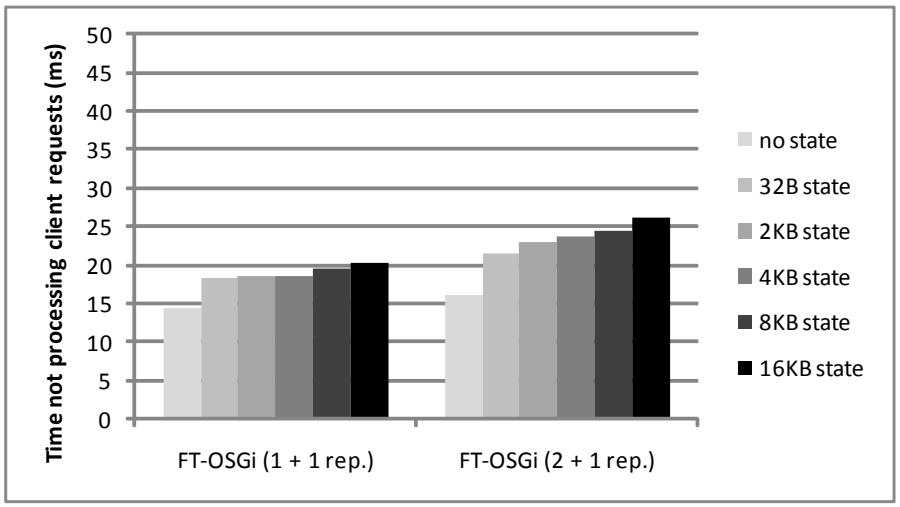



Figure 4.8: Time of group replicas not processing client requests during a new replica joining process.

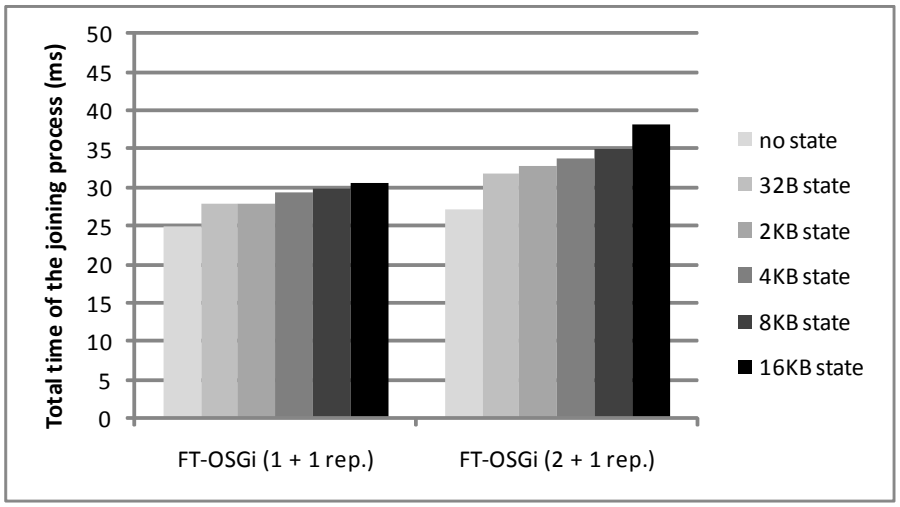



Figure 4.9: Total time of a new replica joining process.

temporary interruption of the data flow to ensure view synchronous properties.

To assess the overhead of these mechanisms we artificially induced the crash of a replica. A non-leader replica is responsible for sending a special multicast control message that causes another target replica to crash. Since every replica receives the special message approximately at the same time, we can measure the time interval between the crash and the moment when a new view, without the crashed replica, is received (and the communication is re-established). Additionally, we have also measure the impact of the crash on the client, by measuring the additional delay induced by the failure in the execution of the client request. We have repeated the same experience 10 times and made an average of the measured results.

The time to install a new group view as soon as a crash has been detected is, on average, $7ms$. The time to detect the failure depends on the Appia configuration. In the standard distribution, Appia is configured to operate over wide-area networks, and timeouts are set conservatively. Therefore, failure detection can take as much as $344ms$. By tuning the system configuration for a LAN setting, we were able to reduce this time to $73ms$. The reader should notice that the failure detection time has little impact on the client experience. In fact, while the replica failure is undetected, the remaining replicas may continue to process (and reply to) client requests. Therefore, the worst case perceived impact on the client is just the time to install a new view, which, on average, has the same order of magnitude of other factors that may delay a remote invocation. As a result, in all the experiments, there were no observable differences from the point of view of remote clients between the runs where failures were induced and failure-free runs.

## 4.8 Discussion

The replication capabilities of FT-OSGi extensions introduce an overhead when compared with a non replicated one. The lazy-passive replication was the strategy with the lowest overhead, followed by the active replication, and then the eager-passive replication. In general, it was observed that the message size has a similar impact in both replicated and non replicated systems. However, adding extra replicas causes an increase in the overhead of the replicated system. The experiments also presented the impact of the filtering modes in active replication: `wait-first` mode has the lowest reliability and overhead; `wait-all` mode has the higher re-

liability and overhead; `wait-majority` mode has a reliability and overhead between the other two modes. The evaluation also presented a direct impact of the services state size in the eager-passive replication and in the new replica join process. Larger service state sizes causes higher overheads in both state update processes. The reader should notice that the lazy-passive replication do not present that kind of overhead when the requests execution times are higher than the state update process times. At last, the replica failure mechanism handling have a little impact on the clients experience.

## Summary

This chapter presented an experimental evaluation of the FT-OSGi extensions. We started by describing a target application area of the FT-OSGi extensions to give a context for the workloads used in the experiments. Then, the evaluation environment and testbed used to perform the experimental evaluation was presented. The chapter presents results for the replication overhead from a client perspective, considering the most relevant scenarios for each replication strategy. The chapter also evaluates the performance of the response filtering modes on active replication, and of the state update effect on passive replication. The overhead of the new replica join procedure and failure handling are also presented. Finally, the results of the experiments were discussed.

# 5 Conclusions

Many OSGi application areas present availability and reliability requirements. For instance, in the domotics area context, the users satisfaction is highly correlated with the availability of the services. Therefore, this dissertation has presented and evaluated FT-OSGi, a set of extensions to the OSGi platform to provide fault tolerance to OSGi applications. The OSGi Service Platform defines a component-based platform for applications written in the Java programming language. The OSGi Framework is composed by several layers and provides the primitives that allow applications to be constructed from small, reusable and collaborative components. At the service layer, the OSGi Framework provides a better decoupling between different modules through the OSGi services. Therefore, the services provide opportunities for increasing the dependability of OSGi based applications in a modular way.

In FT-OSGi, each OSGi service can be replicated and configured to use active or passive replication (eager and lazy) and different services can coexist in the same distributed system, using different replication strategies. Besides the replication strategy, FT-OSGi allows the definition of several configurations, such as: group name, primary views, filtering mode of server replies, and the operation of the group communication service. This flexibility allows applications to use the fault-tolerance options which fit best the application needs.

From an architecture perspective, the FT-OSGi extensions have been implemented using four main components: R-OSGi, jGCS / Appia, FT Service Handler, and FT-Core. Each of this components has different responsibilities and functionalities. The R-OSGi components allows an efficient OSGi services remote invocation. The jGCS / Appia provide an easier group management, configuration and coordination between servers. The FT Service Handler is responsible to provide information about the replicated OSGi services. The FT-Core contains the main implementation of the replication and fault-tolerance features provided. Furthermore, the coherence of server replicas is ensured by three main techniques: leader replica election, new replica join process, and recovering from faults. The FT-OSGi extensions were implemented in

Java and are available as open source software.

The main features of the FT-OSGi prototype have been experimentally evaluated, including each replication strategy, each filtering mode in active replication, the effects of state updates on passive replication, the new replica joining process, and the failure handling. The configurations used in the FT-OSGi evaluation can be applied, for instance, in the domotics area. This area is a good example where the user satisfaction can be increased by the availability and reliability properties. The results show that the overheads induced by the FT-OSGi extensions are acceptable, for this target application.

As future work, it would be interesting to extend FT-OSGi with autonomic management of the group of replicas of each service. This would allow automatic recovery of failed replicas. The work may be also extended to support OSGi services that interact with external applications or persistent storage systems. Finally, the current version of FT-OSGi uses a naive approach to disseminate the state of the objects among the replicas. The current implementation could be improved by propagating only the JVM Heap changes. Such approach would also allow to integrate applications in FT-OSGi without any changes to the source code.

# Bibliography


Ahn, H., H. Oh, & C. O. Sung (2006). Towards reliable osgi framework and applications. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, New York, NY, USA, pp. 1456–1461. ACM.

Amir, Y., C. Danilov, & J. Stanton (2000, June). A low latency, loss tolerant architecture and protocol for wide area group communication. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*.

Anderson, T. & P. Lee (1981). *Fault Tolerance: Principles and Practice*. Prentice/Hall.

Apache Foundation (2009). Apache felix. http://felix.apache.org/.

Avizienis, A. (1995). *The Methodology of N-version Programming*, pp. 23–46. Software Fault Tolerance, Michael R. Lyu, Wiley.

Avizienis, A. & L. Chen (1977, Nov). On the implementation of n-version programming for software fault tolerance during execution. In *Proceedings of the IEEE COMPSAC'77*, pp. 149–155.

Avizienis, A., J.-C. Laprie, & B. Randell (2001, Apr). Fundamental concepts of dependability. In *Research Report No 1145, LAAS-CNRS*.

Baker, M. & M. Sullivan (1992, June). The recovery box: Using fast recovery to provide high availability in the unix environment. In *Proceedings USENIX Summer Conference*, pp. 31–43.

Baldoni, R. & C. Marchetti (2003). Three-tier replication for ft-corba infrastructures. *Softw. Pract. Exper. 33*(8), 767–797.

Bell, M. (2008). *Service-Oriented Modeling (SOA): Service Analysis, Design, and Architecture*. Wiley & Sons.

Benzekri, A. & R. Puigjaner (1992). Titre fault tolerance metrics and evaluation techniques.

Berners-Lee, T., R. Fielding, & L. Masinter (2005). Rfc 3986, uniform resource identifier (uri): Generic syntax.

Carvalho, N., J. Pereira, & L. Rodrigues (2006, Oct). Towards a generic group communication service. In *Proceedings of the 8th International Symposium on Distributed Objects and Applications (DOA)*, Sydney, Australia.

Dobrev, P., D. Famolari, C. Kurzke, & B. Miller (2002, Aug). Device and service discovery in home networks with osgi. *Communications Magazine, IEEE 40*(8), 86–92.

Eclipse Foundation (2009). Equinox. http://www.eclipse.org/equinox/.

Felber, P., B. Grabinato, & R. Guerraoui (1996, Oct). The design of a CORBA group communication service. In *Proceedings of the 15th IEEE Symposium on Reliable Distributed Systems*, Niagara-on-the-Lake, Canada, pp. 150–159.

Fowler, G., Y. Huang, D. Korn, & H. Rao (1993, June). A user-level replicated file system. In *Summer 1993 USENIX Conference*.

Gray, J. (1986). Why do computers stop and what can be done about it? In *Symposium on Reliability in Distributed Software and Database Systems*, pp. 3–12.

Gray, J. & D. Siewiorek (1991, Sep). High-availability computer systems. *Computer 24*(9), 39–48.

Guerraoui, R. & A. Schiper (1997, Apr). Software-based replication for fault tolerance. *Computer 30*(4), 68–74.

Guttman, E. (1999, Jul/Aug). Service location protocol: automatic discovery of ip network services. *Internet Computing, IEEE 3*(4), 71–80.

Herlihy, M. & J. Wing (1990, July). Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems 12*(3), 463–492.

Hsueh, M.-C., T. Tsai, & R. Iyer (1997, Apr). Fault injection techniques and tools. *Computer 30*(4), 75–82.

Huang, Y. & C. Kintala (1993, Jun). Software implemented fault tolerance: Technologies and experience. In *Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers., The Twenty-Third International Symposium on*.

Kaegi, S. & D. Deugo (2008). Modular java web applications. In *Proceedings of the 2008 ACM symposium on Applied computing*, pp. 688–693.

Kaila, L., J. Mikkonen, A.-M. Vainio, & J. Vanhala (2008, May). The ehome - a practical smart home implementation. In *Proceedings of the workshop Pervasive Computing @ Home*, Montpellier, France.

Knopflerfish Project (2009). Knopflerfish. http://www.knopflerfish.org/.

Kriens, P. & B. Hargrave (2004). Listeners considered harmful: The "whiteboard" pattern. Technical report, Technical report, OSGi Alliance.

Li, X. & W. Zhang (2004, May). The design and implementation of home network system using osgi compliant middleware. *Consumer Electronics, IEEE Transactions on 50*(2), 528–534.

Lin, R.-T., C.-S. Hsu, T. Y. Chun, & S.-T. Cheng (2008, 30 2008-Dec. 3). Osgi-based smart home architecture for heterogeneous network. In *Sensing Technology, 2008. ICST 2008. 3rd International Conference on*, pp. 527–532.

Maragkos, D. (2008). Replication and migration of osgi bundles in the virtual osgi framework. Master's thesis, Swiss Federal Institute of Technology Zurich.

Matos, M. & A. Sousa (2008). Dependable distributed osgi environment. In *Proceedings of the 3rd workshop on Middleware for Service Oriented Computing*, pp. 1–6.

Miranda, H., A. Pinto, & L. Rodrigues (2001, Apr). Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, Phoenix, Arizona, pp. 707–710. IEEE.

Narasimhan, P. (1999). *Transparent Fault Tolerance for CORBA*. Ph. D. thesis, Dept. of Electrical and Computer Eng., Univ. of California.

Narasimhan, P., L. Moser, & P. Melliar-Smith (2002, July). Eternal - a component-based framework for transparent fault-tolerant corba. *Software Practice and Experience 32*(8), 771–788.

Nelson, V. (1990, Jul). Fault-tolerant computing: Fundamental concepts. *IEEE Computer 23*(7), 19–25.

Object Management Group (2001, Dec). Fault tolerant corba (final adopted specification). OMG Technical Committee Document formal/01-12-29.

Object Management Group (2004, March). Common object request broker architecture: Core specification, 3.0.3 edition. OMG Technical Committee Document formal/04-03-01.

OSGi Alliance (2007a, Apr). Osgi service platform core specification. http://www.osgi.org/Download/Release4V41.

OSGi Alliance (2007b, Apr). Osgi service platform service compendium. http://www.osgi.org/Download/Release4V41.

Powell, D. (1994, Feb.). Distributed fault tolerance: Lessons from delta-4. In *IEEE Micro*, pp. 36–47.

Pradhan, D. K. (1996). *Fault-Tolerant Computer System Design*. Prentice/Hall.

Randell, B. (1975, June). System structure for software fault tolerance. *IEEE Transactions on Software Engineering SE-1*(2), 220–232.

Randell, B. & J. Xu (1995). *The Evolution of the Recovery Block Concept*, pp. 1–21. Software Fault Tolerance, Michael R. Lyu, Wiley.

Rellermeyer, J., G. Alonso, & T. Roscoe (2007). R-osgi: Distributed applications through software modularization. In *Middleware*, pp. 1–20.

Salas, J., F. Perez-Sorrosal, M. Pati no-Martínez, & R. Jiménez-Peris (2006). Ws-replication: a framework for highly available web services. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pp. 357–366. ACM.

Shwehdi, M. & A. Khan (1996, Jul). A power line data communication interface using spread spectrum technology in home automation. *Power Delivery, IEEE Transactions on 11*(3), 1232–1237.

Thomsen, J. (2006). Osgi-based gateway replication. In *Proceedings of the IADIS Applied Computing Conference 2006*, pp. 123–129.

Torrão, C., N. Carvalho, & L. Rodrigues (2009a, September). Ft-osgi: Extensões à plataforma osgi para tolerância a faltas (artigo curto). In *Actas do Primeiro Simpósio de Informática (Inforum)*, Lisboa, Portugal.

Torrão, C., N. Carvalho, & L. Rodrigues (2009b, November). Ft-osgi: Fault tolerant extensions to the osgi service platform. In *Proceedings of the 11th International Symposium on Distributed Objects and Applications (DOA)*, Algarve, Portugal.

Wilfredo, T. (2000). Software fault tolerance: A tutorial. Technical report, NASA Langley Technical Report Server.

Wu, C.-L., C.-F. Liao, & L.-C. Fu (2007, March). Service-oriented smart-home architecture based on osgi and mobile-agent technology. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on 37*(2), 193–205.