Fault Tolerance in the OSGi Service Platform

Carlos Filipe Lopes Torrão
carlos.torrao@ist.utl.pt

Instituto Superior Técnico

(Advisor: Professor Luís Rodrigues)

Abstract The OSGi Service Platform defines a framework for the deployment of extensible and downloadable Java applications. The framework introduces a management unit, called a bundle, that can be installed, updated, uninstalled, started or stopped without restarting the entire framework. Bundles provide opportunities for increasing the dependability of OSGi based applications in a modular way. This report addresses the problem of augmenting an OSGi framework with fault-tolerant mechanisms.

1 Introduction

The Java programming language is widely used today to build applications that have high availability and reliability requirements. Therefore, it is of utmost importance to design fault-tolerance support for those applications.

Unfortunately, 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 in their article [1] present a solution to this problem, based on the use of a Java modularity technology called OSGi [2]. 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 framework introduces a management unit, called bundle, that can be installed, updated, uninstalled, started or stopped without restarting the entire framework. Bundles provide opportunities for increasing the dependability of OSGi based applications in a modular way, it provides the tools that allow fault-recovery to be applied at the bundle level and not at the level of the entire application. For instance, different fault-tolerance techniques may be applied to each individual bundle, depending of its characteristics. This report addresses the problem of augmenting an OSGi framework with fault-tolerant mechanisms that exploit the modularity features of the framework.

The remaining of this report is organized as follows. Section 2 briefly introduces the goals of our work and its expected outcomes. Section 3 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. Section 4 presents a proposal architecture to add modular fault-tolerance to the OSGi framework and Section 5 describes the approach that will be followed to perform the evaluation of the solution. At last, Section 6 presents the schedule of future work and Section 7 concludes the report.

2 Goals

This work addresses the problem of increasing the dependability of OSGi based systems. In namely:

> *Goals:* This works aims at designing and implementing modular fault-tolerant techniques for OSGi applications.

The work will mainly consider software fault tolerance for OSGi modules, i.e., design solutions that can tolerate software faults in the bundles or that arise from the interaction of the bundles with the OSGi runtime. The solution needs to consider the following criteria:

– Increase of availability and reliability of OSGi modules (known as bundles);
– Provide different levels of the availability and reliability desirable to each bundle, according to the specific characteristics and requirements of each individual bundle;
– Achieve a solution with an acceptable performance.

Thus, instead of supporting a single fault-tolerance strategy, we aim at designing an architecture that supports the application of different fault-tolerant strategies to different bundles. Hence, we expect to achieve the following results:

> *Expected results:* The research will i) define a set of fault-tolerant mechanisms for OSGi bundles; ii) provide a prototype implementation of these mechanisms; iii) implement a demonstration OSGi application that makes use of these mechanisms and, finally; iv) an evaluation of the performance penalties incurred and dependability benefits provided by the approach.

3 Related Work

This section 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. 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.

3.1 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.

Dependability Computing systems are characterized by four properties: functionality, performance, cost and dependability. Following Avizienis et al. [3], 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 instant of time (not an interval of time) that the system is running and responding correctly, i.e., readiness for correct service.
- **Reliability:** Reveals the interval of time that the system is running and responding correctly without any interruption, 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 Failure (MTTF) is the expected time that the system will operate before the first failure occurs [4].

Mean Time to Repair (MTTR) is the average time necessary to repair the system after a failure [4].

Mean Time between Failures (MTBF) is the expected time of a failure occurrence in the system plus the expected necessary time to repair that failure. Therefore, the reliability attribute is proportional to the Mean Time Between Failures (MTBF) [5]. Furthermore, the availability can be defined as the ratio between MTTF and MTBF [4].

Hence, the formulas to calculate the value of reliability and availability attributes are the following:

$$\textbf{Reliability} = MTBF = MTTF + MTTR \tag{1}$$

$$\textbf{Availability} = \frac{MTTF}{MTBF} = \frac{MTTF}{MTTF + MTTR} \tag{2}$$

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.

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.

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.

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.

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 a even better system dependability.

Fault Determinism Faults can be characterized according to its determinism in two distinct classes [5]: 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 [6] consider the name Heisenbug attributed to a transient fault, and they suggest that most software faults in production systems are transient.

Later in this report, these two faults definitions will be essential for the comparison of some fault tolerance techniques.

Replication Techniques Fault tolerance in a system requires some form of redundancy [7], 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* [8], 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 [9]: 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 1 illustrated the processing of one invocation from a client and assuming that no failures occur. The steps are detailed below:

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

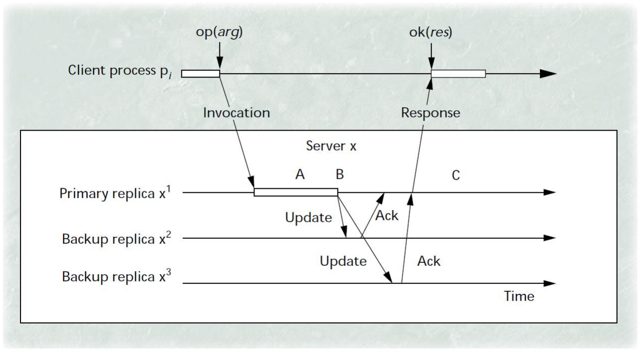

Figure 1. Example of a passive replication interaction (Figure from [9])

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

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)

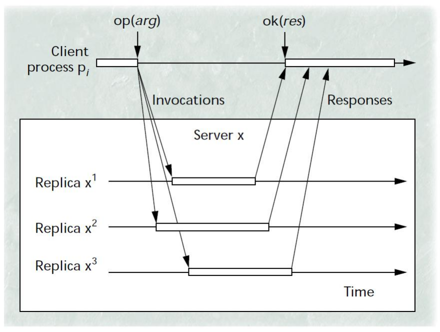

Figure 2. Example of an active replication interaction (Figure from [9])

Hardware / Software Failures

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.

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.

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 [5]. Subsequently it will be presented the two main redundancy approaches related to software modules: single-version and multi-version.

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.

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 the retry of 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 [10].

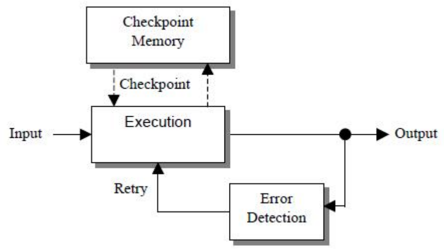

Figure 3. Representation of Checkpoint and Restart Technique (Figure from [11])

In Figure 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 [5].

Process Pairs. A process pair uses the same software module running on separate processors [12] and uses the checkpoint and restart technique, described above, for the recovery. The processors are labeled as primary and secondary (or backup).

In Figure 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

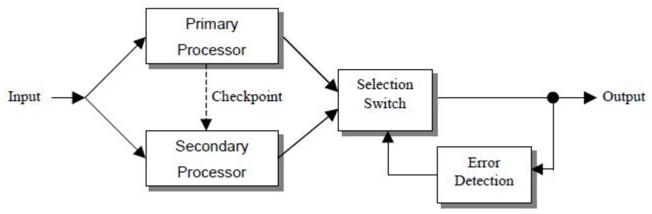

Figure 4. Representation of Process Pairs Technique (Figure from [11])

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.

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 [13]. 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.

Recovery Blocks. The recovery blocks technique [14,15] 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.

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

N-Version Programming. The N-version programming technique [16] is designed to achieve a decision of output correctness from multiple module versions (see Figure 6).

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

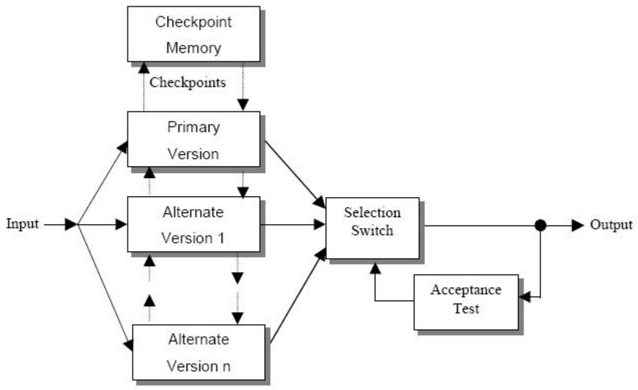

Figure 5. Recovery Blocks Model (Figure from [11])

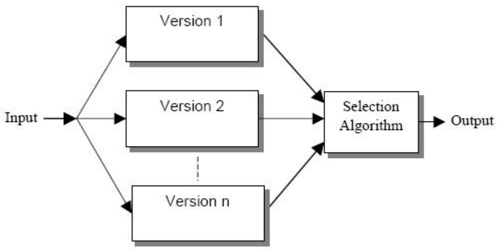

Figure 6. N-Version Programming Model (Figure from [11])

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.

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

Figure 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

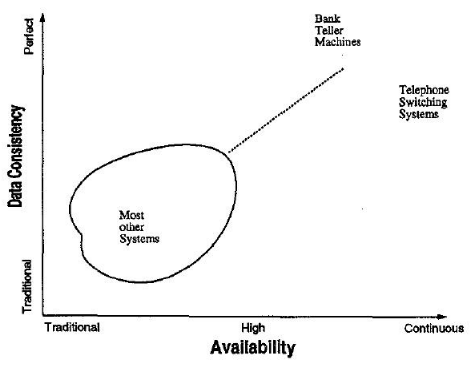

Figure 7. Dimensions of Fault Tolerance (Figure from [17])

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.

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 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:

– **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 [17] with their experience in AT&T decided to define the following five software fault tolerance levels for the applications:

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)

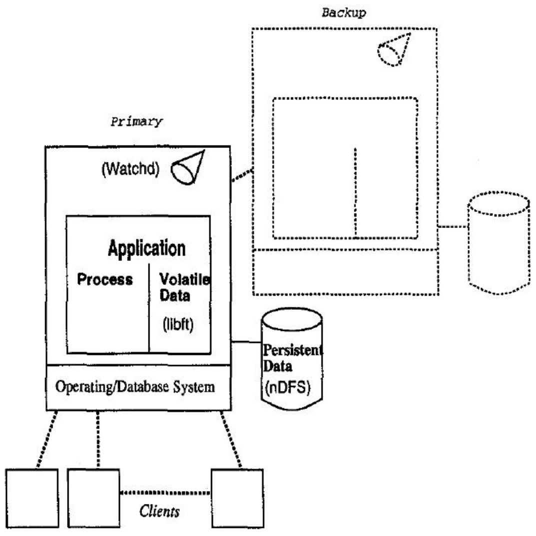

Figure 8. Example of an application in client-server architecture (Figure from [17])

starts from the initial state. Besides that, it is likely that application leaves the persistent data in an incorrect or inconsistent state.

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.

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.

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.

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.

Technologies and Experience Huang and Kintala [17] developed three reusable components possible to integrate with any application, these components are projected to increase the fault tolerance levels (defined previously) of applications.

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.

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 [18]. Watchd and Libft, when combined in an application, can guarantee the Level 2 of the software fault tolerance levels.

nDFS is a multi-dimensional file system based on 3DFS [19] 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.

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. [20], 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.

3.2 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 [21]. 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 described below 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.

CORBA The Common Object Request Broker Architecture [22] (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).

Eternal The Eternal system [23,24] is a component-based framework that provides transparent fault tolerance for CORBA applications. 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.

Ordering of operations 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.

Duplicate operations 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.

Recovery 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.

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

Figure 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 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,

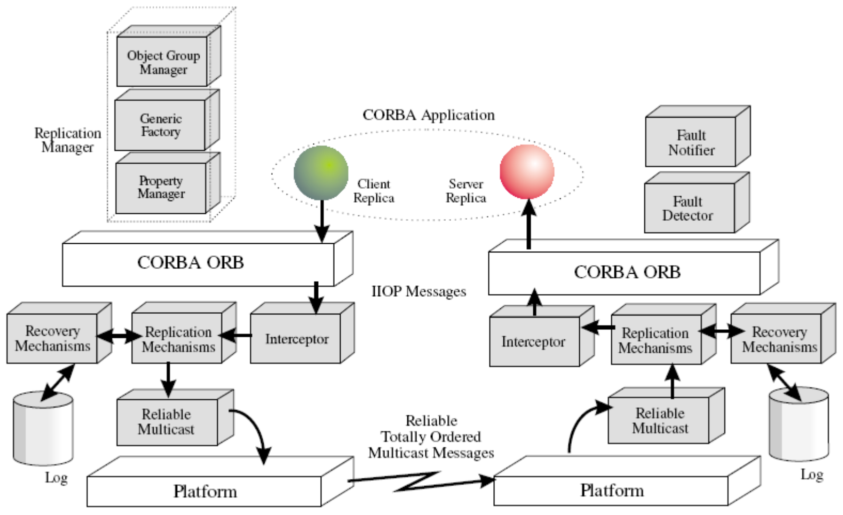

Figure 9. Architectural overview of the Eternal system (Figure from [23])

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 report, 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 is demanded, 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.

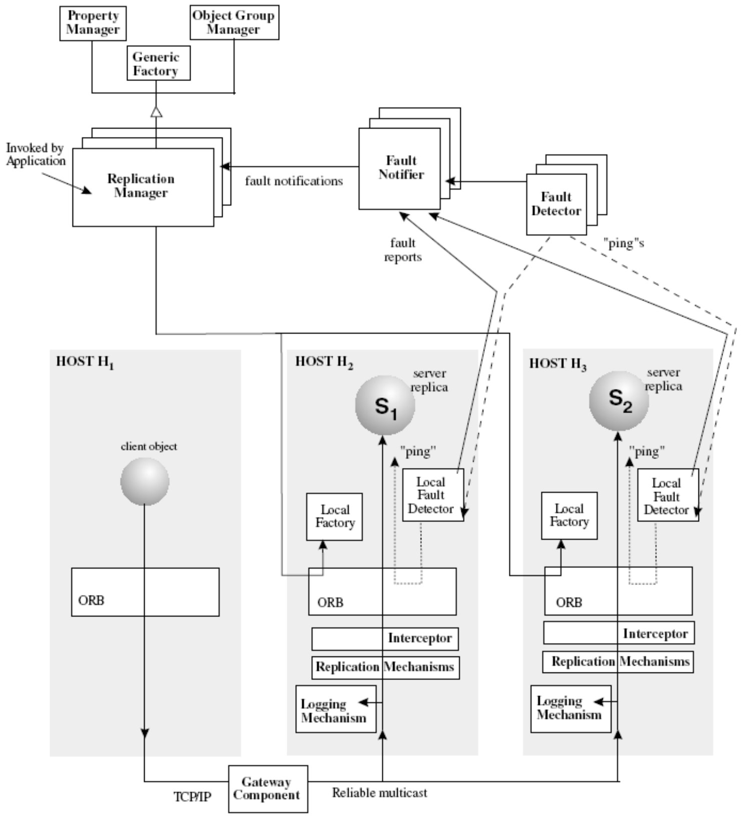

Figure 10. Interaction of Eternal components (Figure from [24])

Figure 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.

3.3 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) [2], 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 report, it is possible to find several OSGi framework implementations of the OSGi specification, such as Apache Felix [25], Eclipse Equinox [26] and Knopflerfish [27].

Architecture The OSGi Framework architecture and functionality is divided in the following major layers, as depicted in Figure 11: Security Layer, Module Layer, Life Cycle Layer, Service Layer, and finally, Actual Services. 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.

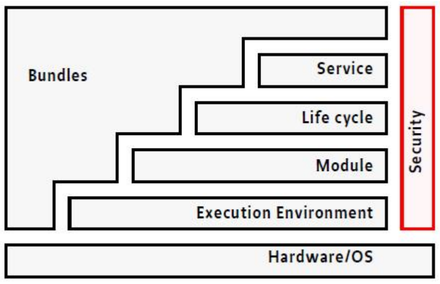

Figure 11. OSGi Framework Layers (Figure from [2])

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.

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 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.

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 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.

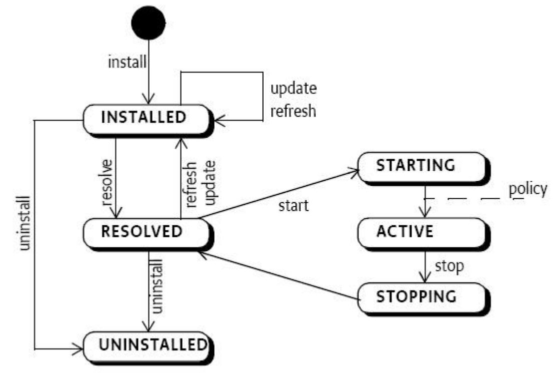

Figure 12. OSGi Bundle State Life Cycle (Figure from [2])

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 (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.

Layer Interactions Figure 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.

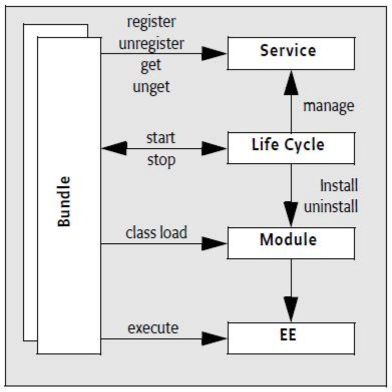

Figure 13. Interactions between the layers (Figure from [2])

3.4 Fault Tolerance in OSGi

OSGi Framework Release 4 does not address the problem of providing fault tolerance support for bundles. 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 [28,29] and a proposals for virtual distributed OSGi [30,31].

Replicated OSGi Gateways Thomsen [28] 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.

What to replicate. To maintain the sub-gateways always replicated is necessary to replicate the bundles (executable code, internal state and persistent data) of the main gateway.

How to replicate. 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.

When to replicate. The executable code is replicated when the buddle suffers an update. The remaining data is replicated periodically or when happens a modification.

Proxy-based OSGi Framework In a similar context, but with focus in the services provided through OSGi Framework, Heejune Ahn et al. [29] 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.

Distributed OSGi Maragkos [30] in his work provides a way to replicate and migrate bundles in a Virtual OSGi Framework [32], which is based in the R-OSGi platform [33].

Remoting-OSGi (R-OSGi) R-OSGi [33] is a distributed middleware platform that can distribute an OSGi application through different nodes running the OSGi Framework. Besides, the R-OSGi layer on top the OSGi is transparent and matches the lightweight design of OSGi. This way, any application prepared to run with OSGi services bundles, can be distributed through R-OSGi.

The approach followed by Rellermeyer et al. to achieve the R-OSGi main goal was through the usage of service proxies and the usage of a centralized service registry. Each service published by a bundle in a specific node, when is necessary in other node (bind) a proxy service is created in the other node. This proxy imitates locally the real service, communicating through the network.

To discover the services available in all nodes, the centralized service registry is used by the R-OSGi.

The Virtual OSGi Framework The concept of Virtual OSGi Framework (VOSGi) [32] is to treat the whole network as a large virtual OSGi Framework, as if it was a simple and not distributed OSGi Framework. This concept uses the R-OSGi logic to achieve that purpose. The main difference of the Virtual OSGi in comparison with R-OSGi is the awareness of the distributed services. In Virtual OSGi the application is not aware of the distribution through several OSGi nodes. Therefore, VOSGi hides the distribution as if it was a simple OSGi Framework running locally in one node.

Replication and Migration of OSGi Bundles in VOSGi As stated previously, Maragkos [30] presents solutions to replicate and migrate OSGi bundles in the Virtual OSGi Framework. The main problem on the migration of a bundle, is related with Java threads, because Java technology does not provide functionality to access thread's state [34]. Therefore, it is necessary to implement a thread serialization mechanism. The implementation by Maragkos of this mechanism is done by the introduction of application code with ASM (bytecode transformer) [35], with the purpose of extracting the internal state of the running thread in specific safe points. With this mechanism implemented, the migration of a bundle through different machines becomes possible.

Dependable Distributed OSGi At the present time, service providers need to provide among the 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. Matos and Sousa [31] 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 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.

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 15).

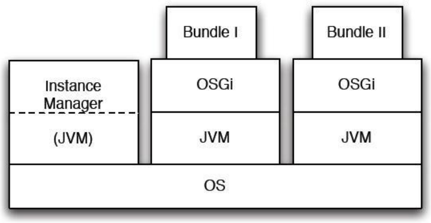

Figure 14. Architecture with multiple OSGi instances on different JVMs (Figure from [31])

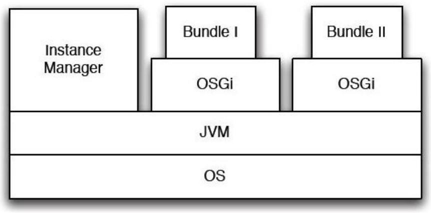

Figure 15. Architecture with multiple OSGi instances with only one JVM (Figure from [31])

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 16). A good feature of this solution is the possibility 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.

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 filesystem 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.

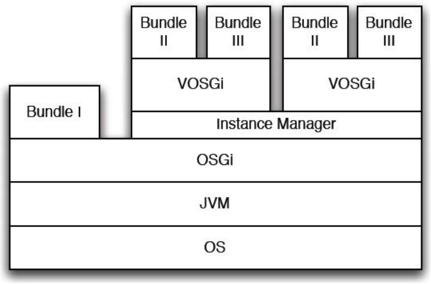

Figure 16. Architecture with multiple OSGi instances inside an OSGi environment (Figure from [31])

4 Proposed Architecture

The aim of this section is to define an architecture that allows to apply modular fault-tolerance to OSGi system.

4.1 Techniques Provided

Our architecture will support the following fault-tolerance techniques (that provide different fault-tolerance levels, using the taxonomy of [17]):

– Stateless Recovery (Level 1 in software fault tolerance)
– Stateful Recovery (Level 2 in software fault tolerance)
– Passive Replication (Level 3 in software fault tolerance)
– Active Replication (Level 3 in software fault tolerance)

The architecture will allow a different mechanism to be applied to different bundles, according to the specific properties of the bundle and the concrete fault-tolerance requirements of the system.

We assume that the developer of the OSGi application will provide in the OSGi bundles manifest information regarding the properties of the bundle such that the appropriate fault-tolerance mechanisms may be selected at deployment time (for instance, Active Replication mechanism requires the bundle to only have deterministic operations).

4.2 Architecture Overview

As described in the related work presenting the OSGi, the OSGi services are based on a publish, find and bind model. This model is supported by the OSGi Framework, by providing a service registry which maintains all the information about the services registered in that framework. These services can be registered only by bundles, and the bundle which has the implementation and registers a service is considered its owner. Therefore, when a bundle is uninstalled, all owned services of that bundle are unregistered.

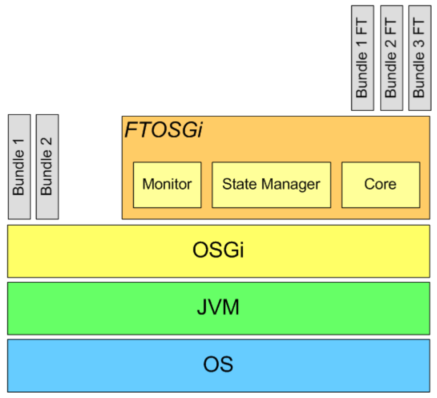

Figure 17. OSGi Bundle Fault Tolerance Solution Architecture

Any registered service can be used by any other bundle running in the framework, and for the other bundles a service is provided as a public interface, which is the only way to interact with that service. Besides, the usage of a service interface is always through the OSGi Framework. For this reason, the fault tolerance support for bundles will be constrained, in our proposal, to the service level of the OSGi Framework, i.e., only the functionality exported as an OSGi service can be provided with fault tolerance support.

Our proposed architecture adds a new layer to the original OSGi Framework architecture. This new layer, which can be implemented as a bundle in the OSGi Framework, will provide dependability services that will allow implementing the four fault-tolerance techniques enumerated before. The resulting architecture is depicted in Figure 17 and is composed of the following layers: the Operating System, Java Virtual Machine, OSGi Framework and FTOSGi layer. The FTOSGi layer is responsible to interact with the bundles that need one of the software fault tolerance mechanisms. Therefore, this layer is divided in three modules: Monitor, State Manager and Core. The Monitor module is responsible to monitor the availability status of the bundles running on top of the FTOSGi layer. Besides, it monitors itself and the other two modules (State Manager and Core) of FTOSGi layer. The State Manager module is responsible for the state capturing of the bundles state. Clearly, this module is only necessary for the Stateful Recovery, Passive Replication and Active Replication mechanisms. In the Stateless Recovery mechanism, this module it is not used at all. The Core module is responsible with the interaction with the below OSGi Framework and the bundles on top of the FTOSGi layer. This module will manage its bundles, keeping all the information necessary to succeed on granting software fault tolerance to them, providing changed bundles to run in the OSGi Framework below.

5 Evaluation Methodology

The evaluation of the proposed architecture and mechanisms will be done experimentally, using a prototype to be developed as future work.

5.1 Metrics

The evaluation will measure essentially properties of dependability, such as availability, reliability and also the performance. Therefore, the crucial metrics to measure the availability and reliability are the following:

- Mean Time to Failure (MTTF)
- Mean Time to Repair (MTTR)
- Mean Time between Failures (MTBF)

Finally, to achieve a performance metric it will be measured the time (in milliseconds) spent by OSGi services to process and respond to requests.

5.2 Methodology

The evaluation will consider different aspects of the proposed mechanisms.

- First we will demonstrate the execution of each mechanism, by developing some simple applications that can benefit from the proposed fault-tolerant techniques and by injecting manually faults in their code to activate the fault recovery/mask mechanisms.
- Then we will assess the performance penalty of the fault-tolerance mechanisms by comparing the performance of OSGi services with and without the mechanisms in place.
- We will also extract some metrics about fault-recovery times, for instance, how fast it takes for a backup to take over the role of a primary when the later fails.
- Finally we will attempt to measure the potential impact of our mechanisms in the dependability of an OSGi system. For that purpose we will attempt to use some fault-injection tools in the running system. Given the timeframe of the work, and the fact that we do not have an easy access to a hardware fault-injection tools, we will attempt to perform software fault-injection, as described in the related work.

6 Scheduling of Future Work

Future work is scheduled as follows:

- January 9 - March 29, 2009: Detailed design and implementation of the proposed architecture, including preliminary tests.
- March 30 - May 3, 2009: Perform the complete experimental evaluation of the results.
- May 4 - May 23, 2009: Write a paper describing the project.
- May 24 - June 15, 2009: Finish the writing of the dissertation.
- June 15, 2009: Deliver the MSc dissertation.

7 Conclusions

Our projects aims at designing and implementing an architecture that provides fault-tolerance to OSGi applications. For that purpose we aim at augmenting the OSGi architecture with a layer that implements a number of different fault-tolerant mechanisms that can be applied in a modular way to different bundles. In this report we have surveyed the relevant related work, sketched the proposed architecture and described our plans to evaluated the results obtained. The detailed specification of the architecture, its implementation and experimental evaluation is left for future work, whose schedule has also been presented.

Acknowledgments

I am grateful to J. Leitão and J. Mocito, for the fruitful discussions and comments during the preparation of this report.

References

1. Kaegi, S., Deugo, D.: Modular java web applications. Proceedings of the 2008 ACM symposium on Applied computing (2008) 688–693
2. Alliance, O.: Osgi service platform core specification (April 2007) http://www.osgi.org/Download/Release4V41.
3. Avizienis, A., Laprie, J.C., Randell, B.: Fundamental concepts of dependability. Research Report No 1145, LAAS-CNRS (Apr 2001)
4. Benzekri, A., Puigjaner, R.: Titre fault tolerance metrics and evaluation techniques (1992)
5. Gray, J.: Why do computers stop and what can be done about it? Proc. of 5th Symposium on Reliability in Distributed Software and Database Systems (Jan 1986) 3–12
6. Gray, J., Siewiorek, D.: High-availability computer systems. Computer **24**(9) (Sep 1991) 39–48
7. Nelson, V.: Fault-tolerant computing: Fundamental concepts. IEEE Computer **23**(7) (Jul 1990) 19–25
8. Herlihy, M., Wing, J.: Linearizability: a correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems **12**(3) (July 1990) 463–492
9. Guerraoui, R., Schiper, A.: Software-based replication for fault tolerance. Computer **30**(4) (Apr 1997) 68–74
10. Anderson, T., Lee, P.: Fault Tolerance: Principles and Practice. Prentice/Hall (1981)
11. Wilfredo, T.: Software fault tolerance: A tutorial. Technical report, NASA Langley Technical Report Server (2000)
12. Pradhan, D.K.: Fault-Tolerant Computer System Design. Prentice/Hall (1996)
13. Avizienis, A., Chen, L.: On the implementation of n-version programming for software fault tolerance during execution. Proceedings of the IEEE COMPSAC'77 (Nov 1977) 149–155
14. Randell, B.: System structure for software fault tolerance. IEEE Transactions on Software Engineering **SE-1**(2) (June 1975) 220–232

15. Randell, B., Xu, J. In: The Evolution of the Recovery Block Concept. Software Fault Tolerance, Michael R. Lyu, Wiley (1995) 1–21
16. Avizienis, A. In: The Methodology of N-version Programming. Software Fault Tolerance, Michael R. Lyu, Wiley (1995) 23–46
17. Huang, Y., Kintala, C.: Software implemented fault tolerance: Technologies and experience. Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers., The Twenty-Third International Symposium on (Jun 1993)
18. Baker, M., Sullivan, M.: The recovery box: Using fast recovery to provide high availability in the unix environment. Proceedings USENIX Summer Conference (June 1992) 31–43
19. Fowler, G., Huang, Y., Korn, D., Rao, H.: A user-level replicated file system. Summer 1993 USENIX Conference (June 1993)
20. Hsueh, M.C., Tsai, T., Iyer, R.: Fault injection techniques and tools. Computer **30**(4) (Apr 1997) 75–82
21. Group, O.M.: Fault tolerant corba (final adopted specification) (Dec 2001) OMG Technical Committee Document formal/01-12-29.
22. Group, O.M.: Common object request broker architecture: Core specification, 3.0.3 edition (March 2004) OMG Technical Committee Document formal/04-03-01.
23. Narasimhan, P.: Transparent Fault Tolerance for CORBA. PhD thesis, Dept. of Electrical and Computer Eng., Univ. of California (1999)
24. Narasimhan, P., Moser, L., Melliar-Smith, P.: Eternal - a component-based framework for transparent fault-tolerant corba. Software Practice and Experience **32**(8) (July 2002) 771–788
25. Foundation, A.: Apache felix http://felix.apache.org/.
26. Foundation, E.: Equinox http://www.eclipse.org/equinox/.
27. Project, K.: Knopflerfish http://www.knopflerfish.org/.
28. Thomsen, J.: Osgi-based gateway replication. Proceedings of the IADIS Applied Computing Conference 2006 (2006) 123–129
29. Ahn, H., Oh, H., Sung, C.: Towards reliable osgi framework and applications. Proceedings of the 2006 ACM symposium on Applied computing (2006) 1456–1461
30. Maragkos, D.: Replication and migration of osgi bundles in the virtual osgi framework. Master's thesis, Swiss Federal Institute of Technology Zurich (2008)
31. Matos, M., Sousa, A.: Dependable distributed osgi environment. Proceedings of the 3rd workshop on Middleware for Service Oriented Computing (2008) 1–6
32. Papageorgiou, D.: The virtual osgi framework. Master's thesis, Swiss Federal Institute of Technology Zurich (2008)
33. Rellermeyer, J., Alonso, G., Roscoe, T.: R-osgi: Distributed applications through software modularization. Middleware (2007) 1–20
34. Zhu, W., Wang, C.L., Lau, F.: Lightweight transparent java thread migration for distributed jvm. Parallel Processing, 2003. Proceedings. 2003 International Conference (Oct 2003) 465–472
35. Bruneton, E., Lenglet, R., Coupaye, T.: Asm: a code manipulation tool to implement adaptable systems. Adaptable and Extensible Component Systems (Nov 2002)