FT-OSGi: Fault Tolerant Extensions to the OSGi Service Platform*

Carlos Torrão, Nuno Carvalho, and Luís Rodrigues

INESC-ID/IST
carlos.torrao@ist.utl.pt, nonius@gsd.inesc-id.pt, ler@ist.utl.pt

Abstract. The OSGi Service Platform defines a framework for the deployment of extensible and downloadable Java applications. Many of the application areas for OSGi have significant dependability requirements. This paper presents and evaluates FT-OSGi, a set of extensions to the OSGi Service Platform that allow to replicate OSGi services. FT-OSGi supports replication of OSGi services, including state-transfer among replicas, supports multiple replication strategies, and allows to apply a different replication strategy to each OSGi service.

1 Introduction

The OSGi Service Platform [1] (Open Services Gateway initiative) defines a component-based platform for applications written in the Java^{TM} programming language. The OSGi framework provides the primitives that allow applications to be constructed from small, reusable and collaborative components. It was developed with several applications in mind, including ambient intelligence, automotive electronics, and mobile computing. Furthermore, its advantages made the technology also appealing to build flexible Web applications [2].

Many of the application areas of OSGi have availability and reliability requirements. For instance, in ambient intelligence applications, reliability issues have been reported as one of the main impairments to user satisfaction [3]. Therefore, it is of utmost importance to design fault-tolerance support for OSGi.

This paper 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 [4], FT-CORBA [5,6,7] and WS-Replication [8], 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, keeping the same properties already provided by the OSGi platform.

A prototype of FT-OSGi was implemented. This prototype leverages on existing tools, such as R-OSGi [9] (a service that supports remote accesses to OSGi

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

R. Meersman, T. Dillon, P. Herrero (Eds.): OTM 2009, Part I, LNCS 5870, pp. 653–670, 2009.

services) and the Appia group communication toolkit [10] (for replica coordination). The resulting FT-OSGi framework can be downloaded from sourceforge[1]. The paper also presents an experimental evaluation of the framework, that measures the overhead induced by the replication mechanisms.

The remaining of the paper is structured as follows. The Section 2 describes the related work. The Section 3 presents the FT-OSGi extensions, describing its architecture, system components, how such components interact and how the proposed extensions are used by applications. The Section 4 presents an evaluation of FT-OSGi. Finally, the Section 5 concludes the paper and points to future research.

2 Related Work

This section makes a brief overview of OSGi and of the fault-tolerance techniques more relevant to our work. Then we overview previous work on fault-tolerant distributed component architectures from which the main ideas were inherited, including Delta-4, FT-CORBA and WS-Replication. Finally, we refer previous research that has addressed specifically the issue of augmenting OSGi with fault-tolerant features.

OSGi. The OSGi Framework forms the core of the OSGi Service Platform [1], 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 a bundle while the framework is running. At the time of writing of this paper, it is possible to find several implementations of the OSGi specification, such as Apache Felix [11], Eclipse Equinox [12] and Knopflerfish [13]. In many application areas, bundles provide services with availability and reliability requirements. For instance, in ambiance intelligence application, a bundle can provide services to control the heating system. Therefore, it is interesting to search for techniques that allow such services to be deployed and replicated in multiple hardware components, such that the service remain available even in the presence of faults.

Fault tolerance. 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. This paper focus on fault-tolerance. Fault tolerance in a system requires some form of redundancy [14], and replication is one of the main techniques to achieve it. There are two main replication techniques [15]: passive and active replication. In passive replication, also known as primary-backup, one replica, called the *primary*, is responsible for processing and respond to all invocations from the clients. The remaining replicas,

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

called the *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. 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 correctness, requests need to be disseminated using a total-order-multicast primitive (also known as atomic multicast). This replication technique has the limitation that the operations processed by the replicas need to be deterministic (thus the name, state-machine).

Object replication. There is a large amount of published work on developing and replicating distributed objects. The Delta-4 [4] architecture was aimed at the development of fault-tolerant distributed systems, offering a set of support services implemented using a group-communication oriented approach. To the authors' knowledge, Delta-4 was one of the first architectures to leverage on group communication and membership technologies to implement several object replication strategies. Delta-4 supported three types of replicated components: *active replication*, *passive replication*, and *semi-active replication* (the later keeps several active replicas but uses a primary-backup approach to make decisions about non-deterministic events).

Arjuna [16] is an object-oriented framework, implemented in C++, that provides tools for the construction of fault-tolerant distributed applications. It supports atomic transactions controlling operations on persistent objects. Arjuna objects can be replicated to obtain high availability. Objects on Arjuna can be replicated either with passive and active replication. Passive replication in Arjuna is implemented on top of a regular Remote Procedure Call (RPC). Failure recovery is done with the help of a persistence storage.

The Common Object Request Broker Architecture (CORBA) [17] is a standard defined by the Object Management Group (OMG), which provides an architecture to support remote object invocations. The main component of the CORBA model is the Object Request Broker (ORB), which act as intermediary in the communication between a client object and a server object, shielding the client from differences in programming languages, platform and physical location. That communication of clients and servers is over the TCP/IP-based Internet Inter-ORB Protocol (IIOP). Several research projects have developed techniques to implement fault-tolerant services in CORBA [5,6,7] eventually leading the design of the FT-CORBA specification [18]. All implementations share the same design principles: they offer fault-tolerance by replicating CORBA components in a transparent manner for the clients. Different replication strategies are typically supported, including active replication and primary-backup. To facilitate inter-replica coordination, the system use some form of group-communication services [19]. To implement recovery mechanisms, CORBA components must be responsible to recover, when 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 [5]). To enable the capture and recover of the application state is necessary that CORBA objects implement Checkpointable interface that contains methods to retrieve (get_state()) and assign (set_state()) the state for that object.

The WS-Replication [8] applies the design principles used in the development of Delta-4 and FT-CORBA to offer replication in the Web Services architecture. It allows client to access replicated services whose consistency is ensured using a group communication toolkit that has been adapted to execute on top of a web-service compliant transport (SOAP).

OSGi replication. To the best of our knowledge, no previous work has addressed the problem of offering fault-tolerance support to OSGI applications in a general and complete manner, although several efforts have implemented one form or another of replication in OSGi. Thomsen [20] presents a solution to eliminate the single point of failure of OSGi-based residential gateways, using a passive replication based technique. However, the solution is specialized for the gateways. In a similar context, but with focus in the services provided through OSGi Framework, Heejune Ahn et al. [21] 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.

3 FT-OSGi

This section presents FT-OSGi, a set of extensions to the OSGi platform to improve the reliability and availability of OSGi applications. This section shows the services provided by such extensions and how their are implemented, describing the several components of FT-OSGi and how these components interact.

3.1 Provided Services

The FT-OSGi provides fault tolerance to OSGi applications. This is done by replicating OSGi services in a set of servers. To access the services, the client application communicates with the set of servers in a transparent way. The services can be stateless or stateful. 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

Table 1. Examples of configuration options for the proposed architecture

Configuration	Replication	Reply	State	Broadcast	Views
A	Active	First	Stateless	Total regular	Partitionable
B	Passive	First	Stateless	Reliable regular	Primary
C	Active	Majority	Stateful	Total uniform	Primary
D	Passive	First	Stateful	Reliable uniform	Primary

lazy-passive. The strategy used for replication of an OSGi service is chosen at configuration time, and different services with different replication strategies can coexist in the same FT-OSGi domain.

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).

When a service is replicated, multiple replies to the same request may be generated. 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. 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 as soon as a majority of similar replies is received. Distribution and replication is 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.

Table 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 System Architecture and Components

Figure 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. The servers maintain the 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

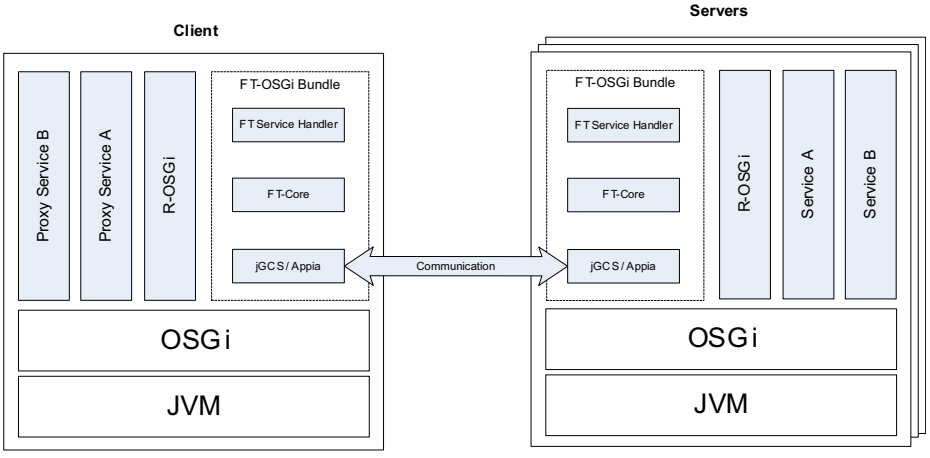


Fig. 1. Architecture of a server and a client

access a service that is installed remotely (in a replicated server), a local proxy is created to simulate the presence of that service.

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 paragraphs:

R-OSGi. R-OSGi [9] 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 an almost transparent way to applications, being possible to run any OSGi application in the R-OSGi platform with only minor changes on stateful services. The R-OSGi layer uses proxies to represent a service that is running in a remote node. To discover the services that are running in remote nodes, R-OSGi uses the *Service Location Protocol* (SLP) [22]. For each service that is advertised by a node in SLP, when another node needs that service, it creates locally a proxy to represent that service. When the application invokes a method in the proxy, that proxy will issue a remote method invocation in a transparent way to applications.

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) [19] that can be configured to use several group communication toolkits, such as Appia [10] or Spread [23]. The prototype presented in this paper uses Appia, 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 in a best effort basis. This service is called *gossip* service and allows the discovery of group members (addresses) from nodes that do not belong to the group.

On top of the previously described services, the following components were built to provide fault tolerance to OSGi services:

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

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 subcomponents that are described next: the *Appia Channel Factory*, the *Client Appia Channel*, the *Server Appia Channel* 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 support the communication between the replicas of the OSGi service and communication between the client and the group of replicas. The client is outside the group and uses the open group functionality supported by Appia. The communication between replicas uses view synchrony (with total order in the case of active replication). For each one of these communication types, 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 in the *Server Appia Channel* component. The communication channel between clients and service replicas is created when some client needs to access a replicated service and is implemented in the *Client Appia Channel* component. Finally, the *Replication Mechanisms* component implements the replication protocols used in FT-OSGi (active and 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, this components uses the membership service provided by the GCS.

3.3 Replication Strategies

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

Active replication. This replication strategy follows the approach of standard active replication [15], 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 by the same global order. One limitation of this replication strategy is that it can only be applied to deterministic services.

Eager passive replication. In this case only one replica, the primary, deals with invocations from the clients [15]. 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.

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, but is faster and many applications do not require strong guarantees.

3.4 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.

Leader election. The replication protocols implemented in the *Replication Mechanisms* component need a mechanism for leader election for several reasons. In the case of passive replication, leader election is necessary to choose the primary replica, executing the requests and disseminating the updates to the other replicas. In the case of active replication, 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 identification that also belonged to the previous view.

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 number of new replicas joining in the view. The state transfer also contains the services configurations for validity check purposes. When a joining replica receives the state message, it validates the service configurations, and updates its own state, it broadcasts an acknowledgment to all the members of the group. Finally, when all the group members receive a number of acknowledgments equal to the number of 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, the remaining replicas will decrement, for each new joined replica that fails, the number of expected acknowledgments. 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.

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 [22] protocol 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 *Server Appia Channel* 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

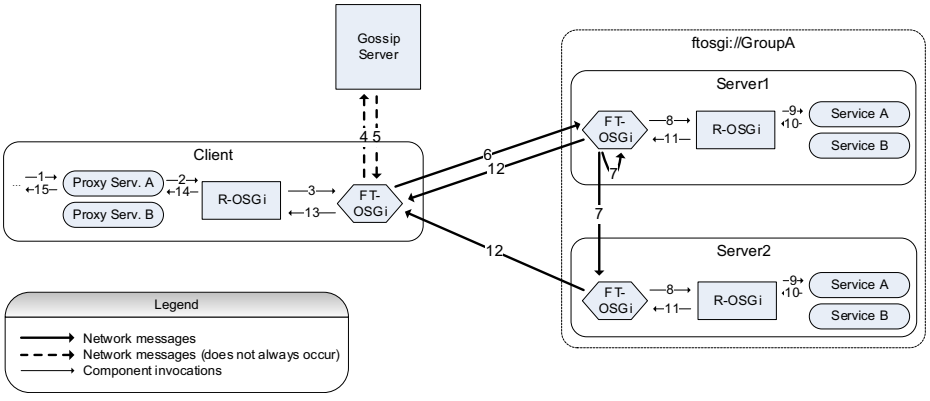


Fig. 2. Interaction between components when using active replication

also create an instance of *Server Appia Channel*, *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 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 *Client Appia Channel* 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.

At this stage, replicas are deployed by scripts on behalf of the system administrator. As future work we plan to implement service factories that can create replicas on demand and provide support to components that perform the autonomic management of the membership. The next section describes how the several FT-OSGi components interact when a client invokes a replicated service.

3.6 Interaction between Components

The interaction between the system components depends on the replication strategy used by the service. For simplicity reasons, we will only illustrate the operation of active replication. The operation of passive replication is very similar, with the obvious differences of the replication protocol: only the primary executes the request, if the service is stateful, the primary reliable broadcasts a state update to all the replicas, and only the primary replies to the client.

The Figure 2 depicts the interaction between the several components of the FT-OSGi extensions. The client wants to invoke a method provided by Service A, which is replicated in Group A, 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 a Remote Method Invocation 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 *Client Appia Channel* component (steps 3 and 6). If the client does not have cached at least one address of the members of Group A, it queries the *gossip* service (steps 4 and 5). This request to the *gossip* service is also done periodically, in order to maintain the cache updated and is resent until it is successfully received by one of the servers. When one of the servers receives the client request, it atomically broadcasts the request to all the servers on Group A, using the *Server Appia Channel* 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 4 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 (steps 14 and 15).

3.7 Programing Example

This section illustrates how a client application and a service are implemented in the FT-OSGi architecture. We will start by showing how to implement and configure a service. The Listing 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 7). After implementing the service, it must be configured and registered in the OSGi platform. This is done 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 12 to 18). The registration of the service in the OSGi platform makes the service available to its clients and is done after the configuration process (line 19).

The same listing also shown 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 27 to 32). This `FTFramework` service is responsible to abstract the interaction with the SLP. Using that service, the application obtains the address `ftosgi://GroupA` (line 33), 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 34 and 35), 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 40), which follows the invocation procedure of an actively replicated service, like it was described in the Section 3.6.

Listing 1. Example code

```
// server code
public class HelloServiceImpl implements HelloService {
 public String speak() {
  return "Hello World!";
 }
 public String yell() { return ("Hello World!".toUpperCase().concat("!!!")); }
}
public class Activator implements BundleActivator {
 private HelloService service;
 public void start(BundleContext context) throws Exception {
  service = new HelloServiceImpl();
  Dictionary<Object, Object> properties = new Hashtable<Object, Object>();
  properties.put(RemoteOSGiService.R_OSGi_REGISTRATION, Boolean.TRUE);
  properties.put(FTServiceTypes.FT_ROSGi_REGISTRATION, Boolean.TRUE);
  properties.put(FTServiceTypes.FT_ROSGi_FT_SERVICE_ID, "HelloService");
  properties.put(FTServiceTypes.FT_ROSGi_FT_TYPE, FTTypes.ACTIVE_STATELESS);
  properties.put(FTServiceTypes.FT_ROSGi_FT_GROUPNAME, "GroupA");
  properties.put(FTServiceTypes.FT_ROSGi_PRIMARY_VIEW, Boolean.TRUE);
  context.registerService(HelloService.class.getName(), service, properties);
 }
 public void stop(BundleContext context) throws Exception { service = null; }
}

// client code
public class Activator implements BundleActivator {
 public void start(BundleContext context) throws Exception {
  final ServiceReference ftRef = context.getServiceReference(FTFramework.class.getName());
  if (ftRef == null) {
    System.out.println("No FTFramework found!");
    return;
  }
  FTFramework ftFramework =(FTFramework)context.getService(ftRef);
  URI helloURI = ftFramework.getFTServiceURI(HelloService.class.getName());
  HelloService helloService =
   (HelloService)ftFramework.getFTService(HelloService.class.getName(), helloURI);
  if(helloService == null) {
   System.out.println("No HelloService found!");
   return;
  } else {
    System.out.println("Response: " + helloService.speak()); // Can start use service
  }
 }
 public void stop(BundleContext context) throws Exception {}
}
```

3.8 Some Relevant OSGi Implementation Details

This section focus in presenting the main issues that emerge by the replication of an OSGi service.

OSGi Service ID. Each OSGi service registered in an OSGi instance has a service id (`SID`) attributed by the OSGi framework itself. This SID is a `Long` object and identifies the service in an unique manner. R-OSGi uses this SID to identify remote services through different nodes since that SID is unique in each node. By extending that concept with an replicated service through several different nodes, the SID does not identify uniquely each service replica in all nodes, because the SID can be attributed differently in each node by the local OSGi instance for the replicated services. To solve 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 was defined as a `String` object to let the developer choosing a more descriptive id, allowing unlimited and name space ids. 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.

Filtering Replicated Events. The OSGi event mechanisms support the publish-subscribe paradigm. When replicating services, different replicas may publish

multiple copies of the same event. These copies need to be filtered, to preserve the semantics of a non-replicated service. FT-OSGi addresses this issue using a similar approach as for filtering 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.

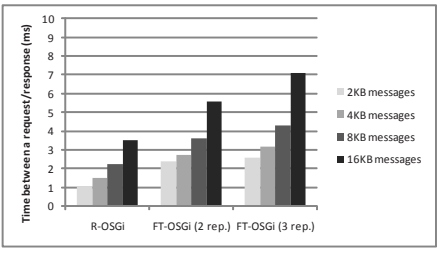
4 Evaluation

This section presents an evaluation of the FT-OSGi extensions, that will focus on the overhead introduced by replication. For these tests, an OSGi service with a set of methods was built, that (*i*) receive parameters and return objects with different sizes, generating requests with a growing message size, and (*ii*) have different processing times. The response time on FT-OSGi was measured with several replication strategies and compared it with R-OSGi, which is distributed but not replicated.

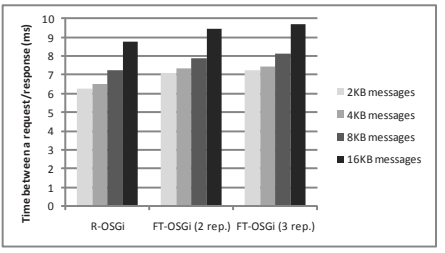
4.1 Environment

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 *gossip* service, which is a light process that does not affect the processing time of the FT-OSGi server. The client machine has one Intel Pentium 4 @ 2.80 Ghz (with Hyper-threading) 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.

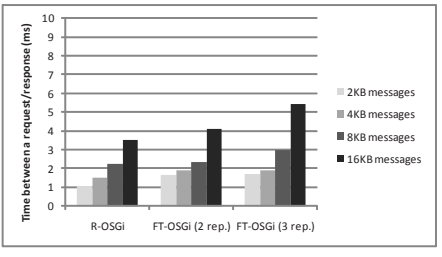
All the tests measure the time (in milliseconds) between the request and the reply of a method invocation on a OSGi service. In R-OSGi, the test was performed by invoking a service between a client and a server application. In FT-OSGi, different configurations were considered. The client issues a request to a group of 2 and 3 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.



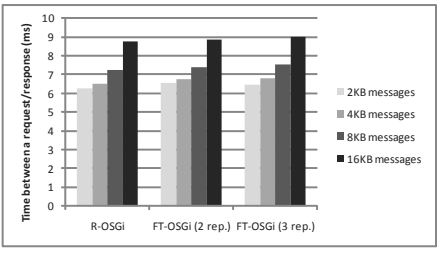
(a) Eager passive, no execution time.



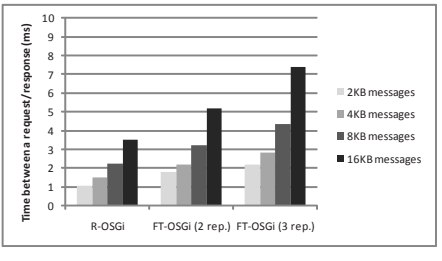
(b) Eager passive, execution time of $5ms$.



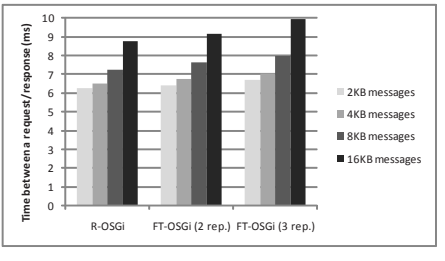
(c) Lazy passive, no execution time.



(d) Lazy passive, execution time of $5ms$.



(e) Active, no execution time.



(f) Active, execution time of $5ms$.

Fig. 3. Replication overhead on different replication strategies

4.2 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 3 were executed using the *wait-first* reply filtering mode. The tests for both eager and lazy passive replication strategies were executed using a stateful service with 32 bytes of state. It was measured the response time with message sizes of 2 KBytes, 4 KBytes, 8 KBytes and 16 KBytes. Figure 3 shows the overhead of replication on active and passive replication. In the tests with no execution time, 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 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 lower overhead (Figure 3(c)). 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. The tests presented in Figure 3 with an execution time of the invoked method of $5ms$, show that the overhead is smaller in all replication strategies, meaning that the execution time dominate the overhead of replication.

4.3 Response Filtering Modes on Active Replication

The Figure 4 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 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.

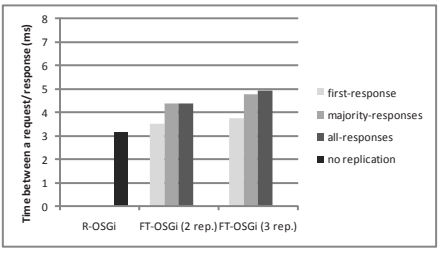
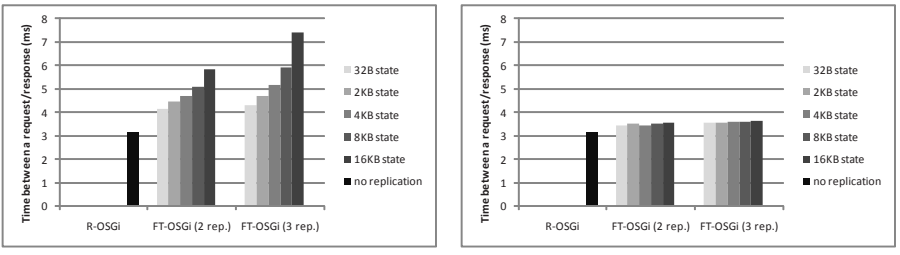


Fig. 4. Response time on active replication with the 3 filtering modes

4.4 Effect of State Updates on Passive Replication

Figure 5 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 5(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 5(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.

Fig. 5. Passive replication with different state sizes

4.5 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 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 almost 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 time 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 and, on average, much smaller than that, and in 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.

5 Conclusions and Future Work

This paper presents and evaluates FT-OSGi, a set of extensions to the OSGi platform to provide fault tolerance to OSGi applications. In FT-OSGi, each service can be configured to use active or passive replication (eager and lazy) and different services can coexist in the same distributed system, using different replication strategies. These extensions where implemented in Java and are available as open source software.

As future work we plan to implement the autonomic management of the group of replicas of each service. This will allow automatic recovery of failed replicas. We also plan to extend this work 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. We intent to improve the current implementation by propagating only the JVM Heap changes. This approach will also allow that OSGi application can be integrated in FT-OSGi without needing any changes.

Acknowledgments. The authors wish to thank João Leitão by his comments to preliminary versions of this paper.

References

1. OSGi Alliance: Osgi service platform core specification (April 2007), `http://www.osgi.org/Download/Release4V41`
2. Spring Source: Spring Dynamic Modules for OSGi (2009), `http://www.springsource.org/osgi`
3. Kaila, L., Mikkonen, J., Vainio, A.M., Vanhala, J.: The ehome - a practical smart home implementation. In: Proceedings of the workshop Pervasive Computing @ Home, Sydney, Australia (May 2008)
4. Powell, D.: Distributed Fault Tolerance: Lessons from Delta-4. IEEE Micro", 36–47 (Feburary 1994)
5. Narasimhan, P., Moser, L., Melliar-Smith, P.: Eternal - a component-based framework for transparent fault-tolerant corba. Software Practice and Experience 32(8), 771–788 (2002)
6. Felber, P., Grabinato, B., Guerraoui, R.: The Design of a CORBA Group Communication Service. In: Proceedings of the 15th IEEE SRDS, Niagara-on-the-Lake, Canada, pp. 150–159 (October 1996)
7. Baldoni, R., Marchetti, C.: Three-tier replication for ft-corba infrastructures. Softw. Pract. Exper. 33(8), 767–797 (2003)
8. Salas, J., Perez-Sorrosal, F., no-Martínez, M.P., Jiménez-Peris, R.: Ws-replication: a framework for highly available web services. In: WWW 2006: Proc. of the 15th int. conference on World Wide Web, pp. 357–366. ACM, New York (2006)

9. Rellermeyer, J., Alonso, G., Roscoe, T.: R-osgi: Distributed applications through software modularization. Middleware, 1–20 (2007)
10. Miranda, H., Pinto, A., Rodrigues, L.: Appia, a flexible protocol kernel supporting multiple coordinated channels. In: Proceedings of the 21st International Conference on Distributed Computing Systems, pp. 707–710. IEEE, Los Alamitos (2001)
11. Apache Foundation: Apache felix, `http://felix.apache.org/`
12. Eclipse Foundation: Equinox, `http://www.eclipse.org/equinox/`
13. Knopflerfish Project: Knopflerfish, `http://www.knopflerfish.org/`
14. Nelson, V.P.: Fault-tolerant computing: Fundamental concepts. IEEE Computer 23(7), 19–25 (1990)
15. Guerraoui, R., Schiper, A.: Software-based replication for fault tolerance. Computer 30(4), 68–74 (1997)
16. Parrington, G.D., Shrivastava, S.K., Wheater, S.M., Little, M.C.: The design and implementation of arjuna. Technical report (1994)
17. Object Management Group: Corba: Core specification, 3.0.3 ed, OMG Technical Committee Document formal/04-03-01 (March 2004)
18. Narasimhan, P.: Transparent Fault Tolerance for CORBA. PhD thesis, Dept. of Electrical and Computer Eng., Univ. of California (1999)
19. Carvalho, N., Pereira, J., Rodrigues, L.: Towards a generic group communication service. In: Proc. of the 8th Int. Sym. on Distributed Objects and Applications (DOA), Montpellier, France (October 2006)
20. Thomsen, J.: Osgi-based gateway replication. In: Proceedings of the IADIS Applied Computing Conference 2006, pp. 123–129 (2006)
21. Ahn, H., Oh, H., Sung, C.: Towards reliable osgi framework and applications. In: Proceedings of the 2006 ACM symposium on Applied computing, pp. 1456–1461 (2006)
22. Guttman, E.: Service location protocol: automatic discovery of ip network services. Internet Computing, IEEE 3(4), 71–80 (1999)
23. Amir, Y., Danilov, C., Stanton, J.: 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 (June 2000)