Livrable Selware SP4

Lot 3, November 27, 2008, 18:58

Self-Repairing a JMS-based Application with Selfware version 1.0

F. Boyer (INRIA)
S. Sicard (INRIA)
N. De Palma (INRIA)
A. Freussinet (ScalAgent)

Contents

1 Objectives

The Selfware project aims at providing an infrastructure for developing autonomic management software. An important aspect of this infrastructure is the adoption of an architecture-based control approach as described in the SP1-L1 document, where autonomic functions are provided as control loops that regulate the system through introspecting and adapting its software architecture.

self-repair (also called self-healing) is one of the major autonomic functions as defined by IBM[2], which deals with automatic discovery and correction of faults in software systems. Hence a self-repair system must be able to detect problems, diagnose them and recover from faults that might cause some of its parts to malfunction by applying a corrective action, e.g., finding an alternate way of using ressources or reconfiguring the system to keep it working.

The design and implementation principles of the Selfware self-repair function have been described in the SP1-L2 document. The objective of this document is to report on our experiment in applying the self-repair function on a legacy application that is build on top of a JMS (Java Messaging Service) standard platform[4]. This application is a Train Station Application used by a French company. The aim of this application is to allow human administrators to monitor and to manage remote equipments such as escalators, fans, sound, telephony, or video surveillance systems.

The following of this document is organized as follow. Section 2 recalls the main principles of the JMS specification. In our experiment, we used the Joram implementation of the JMS specification, that is described in section 3. Then, section 4 explains how the software elements composing a Joram application have been encapsulated within Selfware manageable components. Based on this encapsulation, we show how a Joram application can be deployed autonomically with Selfware. Once deployed, a Joram application is automatically put under the control of the self-repair autonomic manager. Section 5 describes how the self-repair function behaves in the context of this legacy application.

2 JMS

2.1 A JMS Infrastructure Overview

A JMS plateform is formed of one or several interconnected servers (also called JMS Providers) that can run on remote nodes. A JMS server is a Java process (a JVM) providing messaging functionnalities and hosting message destinations. A JMS server can host 2 types of destination: the queues and the topics (detailed below). A JMS client is a Java process (a JVM) using the messaging functionnalities provided by a server through JMS interfaces. The client needs to have previously connected to a server to execute these operations.

The figure 1 shows a simple JMS architecture. The plateform is constituted of three servers (Server0, Server1 et Server2) and two clients. One of the clients is connected to Server0 and the other to Server2. As these servers host destinations, clients are able to send and receive messages using them. Two types of destinations are provided: the queues and the topics.

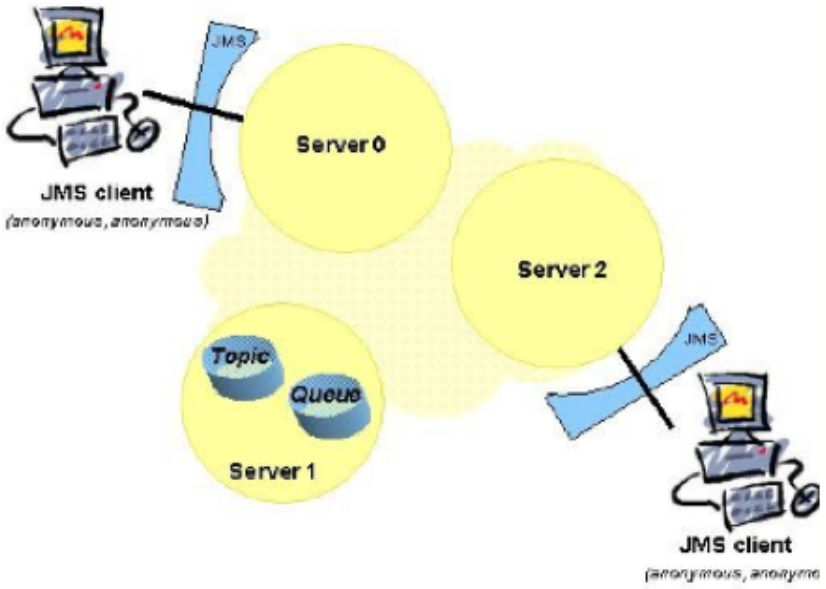

Figure 1: JMS architecture example

2.2 The Queues

The queue follows a producer/consumer pattern. Its principle is the following: while a message has not been withdrawn by a client, it stays in the queue. So, there are no lost messages. In the same way, if a client tries to get a message on an empty queue, he waits until one arrives. A queue also has a special attribute, NbMaxMsg, which is the max number of undelivered messages it can keep. If this limit is reached and more messages are sent to the queue, these ones are immediately sent to a special destination, the Dead Message Queue. The most simple use case of this kind of destination is to have one sender putting messages in the queue and one receiver getting them.

2.3 The Topic

The topic follows a Publish/Subscribe pattern. It corresponds to a destination for which, all clients that have subscribe at time t, receive messages published after this time. Messages are not saved and it is also necessary to subscribe to a topic to get its messages. Messaging with a queue can be considered as a Point-To-Point communication whereas messaging with a topic corresponds to information diffusion.

2.4 The Connection Factories

In order for client to connect and use destinations, the JMS specification defines a set of connection factories. Indeed, to be able to send or receive through a destination, a client needs a reference towards this destination and a connection factory. The connection factory allows him to create a session on the destination and then, to get or send messages if it

is authorized to do it. Figure 2 shows how a client can use a destination. To access the destination, it has first to create a connection using the connection factory.

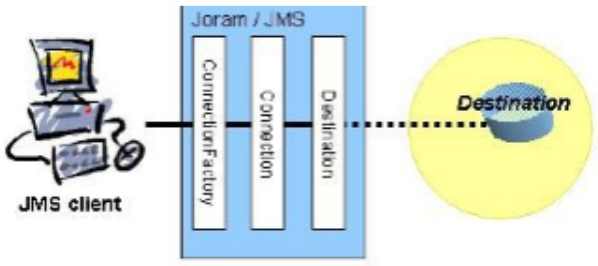

Figure 2: A client using a destination

2.5 The JNDI Registry

A JMS platform is ditributed between computers and in most cases, clients and servers are on remote nodes. That's why the servers use a registry, often a JNDI registry, to bind objects (destinations) to names. Thus, a server binds all its destinations and connection factories in the JNDI registry. Then, a client can look for them and create a connection on the destination of its choice.

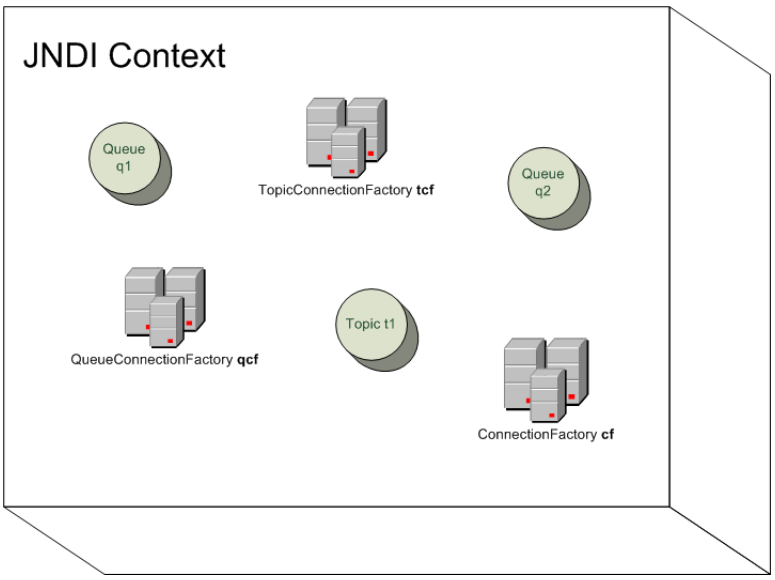

Figure 3: JNDI registry with JMS objects

3 Joram

JORAM is a 100% Java implementation of *JMS* (Java Message Service). It provides access to a MOM (Message Oriented Middleware), built on top of the agents ditributed platform called ScalAgent (see http://www.scalagent.com). It is an *Open Source* software published under *LGPL* License (Lesser GNU Public License).

3.1 Joram Infrastructure Overview

Joram servers are grouped inside domains in which they can communicate and collaborate to manage distributed architectures. Domains are transparent for users. They have been created to gather a set of servers of an architecture that use the same communication protocol.

3.2 Access Rights on Destinations

To bring security, Joram defines the concept of user. It allows only authorized clients to send/publish or get messages on destinations. Thus, a server defines a set of users and the access rights on destinations are managed according to these users. A user is defined by a login and a password.

There is a special user which represents the administrator. This one has the right to modify the architecture of the Joram plateform by adding/removing destinations for example. It is necessary, before calling a function which modifies the architecture, to establish an administrator connection to the server.

As access rights to destinations are managed according to users, a client has to specify a login and a password of a user which has the rigth to access this destination. Moreover, a destination can be freely readable or writable.

3.3 Server Configuration for a Base Architecture

Let's consider the example of a Joram server with a minimum configuration: one queue and one topic. Figure 4 illustrates this architecture.

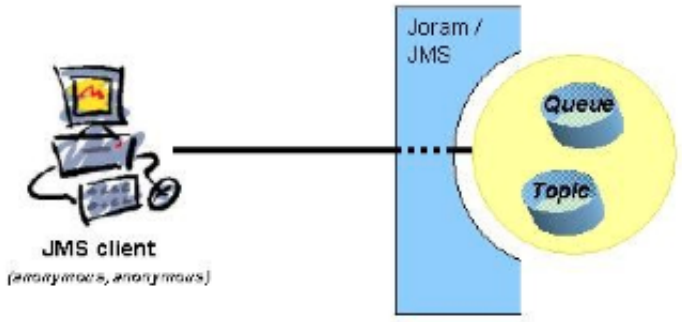

Figure 4: A Joram base architecture

To deploy and activate this architecture, three steps are needed: define an *a3servers.xml* file that sets up the configuration of the Joram servers, start the Joram servers, and execute an administration Java Class.

The *a3servers.xml* file describes statically the several domains and their servers. For each server, a set of services can be defined. For example, a connection manager service that allows an administrator to connect to the server to modify its configuration or a JNDI service that specifies that this server hosts a JNDI registry.

```
<?xml version="1.0"?>
<config>
  <property name="Transaction" value="fr.dyade.aaa.util.NullTransaction"/>

  <server id="0" name="S0" hostname="localhost">
    <service class="org.objectweb.joram.mom.proxies.ConnectionManager"
             args="root root"/>
    <service class="org.objectweb.joram.mom.proxies.tcp.TcpProxyService"
             args="16010"/>
    <service class="fr.dyade.aaa.jndi2.server.JndiServer" args="16400"/>
  </server>
</config>
```

The administration Java class first makes a connection to a given server using the static method *connect* of the Joram class *AdminModule*. It takes as parameters the administator login and password. The connection to *AdminModule* allows to manage JMS and Joram objects of the server (queues, topics, user...) and to manage the running Joram plateform by adding/removing domains and servers.

```
public class ArchiBaseAdmin
{
  public static void main(String[] args) throws Exception
  {
    // Administrator Connection
    AdminModule.connect("root", "root", 60);

    // Destinations creation
    Queue queue = (Queue) Queue.create("queue");
    Topic topic = (Topic) Topic.create("topic");

    // Users creation (for access rigths management)
    User user = User.create("anonymous", "anonymous");

    // Set access rights
    queue.setFreeReading();
    topic.setFreeReading();
    queue.setFreeWriting();
    topic.setFreeWriting();

    // Connection factories creation
    javax.jms.ConnectionFactory cf =
      TcpConnectionFactory.create("localhost", 16010);
    javax.jms.QueueConnectionFactory qcf =
      QueueTcpConnectionFactory.create("localhost", 16010);
    javax.jms.TopicConnectionFactory tcf =
      TopicTcpConnectionFactory.create("localhost", 16010);

    // Bindings in JNDI
    javax.naming.Context jndiCtx = new javax.naming.InitialContext();
    jndiCtx.bind("cf", cf);
    jndiCtx.bind("qcf", qcf);
    jndiCtx.bind("tcf", tcf);
    jndiCtx.bind("queue", queue);
    jndiCtx.bind("topic", topic);
    jndiCtx.close();

    // Administrator disconnection
    AdminModule.disconnect();
  }
}
```

3.4 Use Case Scenario

The workflow of figure 5 shows the actions that a JMS client has to do to use a destination.

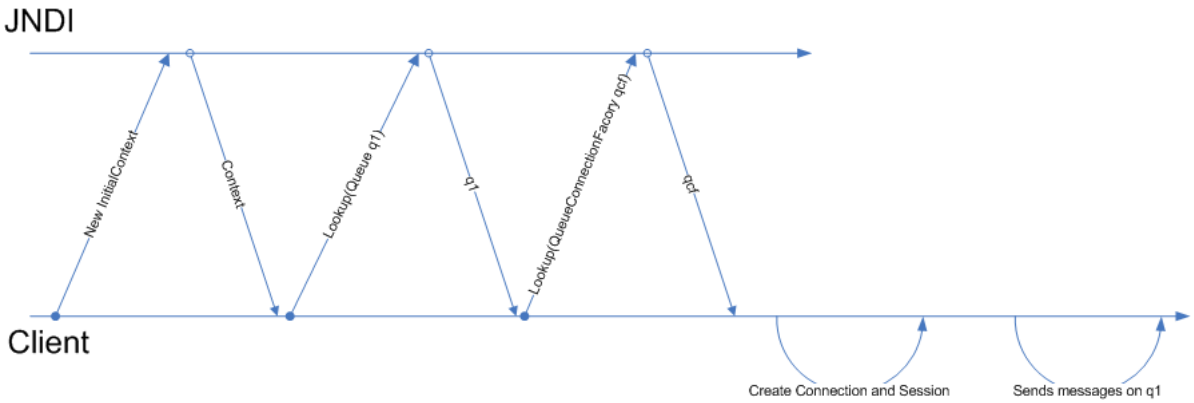

Figure 5: User worflow

First of all, it has to get a reference on the JNDI registry where all the JMS objects are bound. Then, it can get from this registry the destination and connection factory it wants to use. After that, it can start a connection, then create a session and finally send and receive messages.

3.5 Advanced Destinations

More advanced destination features are provided by Joram, such as hierarchical topics used for organizing topics into a hierarchical structure, and cluster queues used for high availability purposes. The description of these features is out of bound of this paper.

4 Autonomic Management of Joram with Selfware

To be managed by Selfware, the software elements of a Joram Application (server, queue, topic, user, etc.) have first to be encapsulated within Selfware manageable components, in order to provide a uniform management interface. In Selfware, we choose the Fractal component model [3] to build these components. The following sections describe how the main Joram software elements (a JNDI element, a Joram server, a Cluster Queue, and Destination and Users) are encapsulated within fractal components.

4.1 JNDI Registry

A Joram domain, associated to a JNDI server, is represented by a simple Fractal composite component. All servers and cluster queues components are bound to this JNDI component because they need a direct access to the registry to bind destinations and connection factories. The JNDI component is directly bound in the Fractal RMI registry with the name *JndiRegistry*. We also suppose that there is only one JNDI registry per Joram

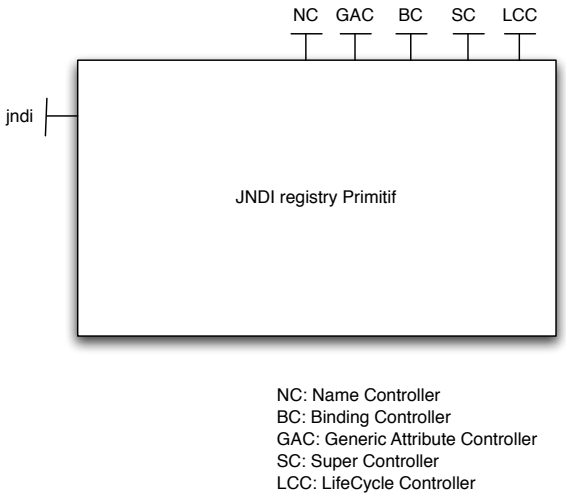

Figure 6: Fractal primitive JNDI registry component

architecture deployed with Jade. The JNDI interface listed below presents methods that provide the JNDI server host and port.

```
public interface JndiInterface {
        public String getHost();
        public String getPort();
}
```

4.2 Joram Servers

A Joram server is represented by a Fractal composite component. Its content controller has been overridden to add some specific treatments when we add or remove a queue, a topic or a user. For example, if we choose to add a queue, the *addFcSubComponent* method starts the queue component and then calls the setProperties method of its queue interface (this is in the setProperties method that the queue is bound in the JNDI registry). This composite has a client interface named *jndi* that is bound to the component wrapping the JNDI registry. So, the composite keeps a reference to the JNDI and if it has to bind an object, can do it easily.

If we restart the JNDI server on an other host during execution (in a case of repair), we just have to update the binding to make the server able to bind objects. With this solution, the user doesn't need to give the host and port of JNDI server as parameters when he starts a server. The Selfware command to create a server introspects the Fractal RMI registry to find the object named *JndiRegistry* and binds the server composite to it. Moreover, this composite life cycle has been overridden to link the Fractal and legacy life cycles. So, starting the component at Fractal level leads to starting the Joram server.

In addition, starting a Joram server requires specific actions. There must be, in the

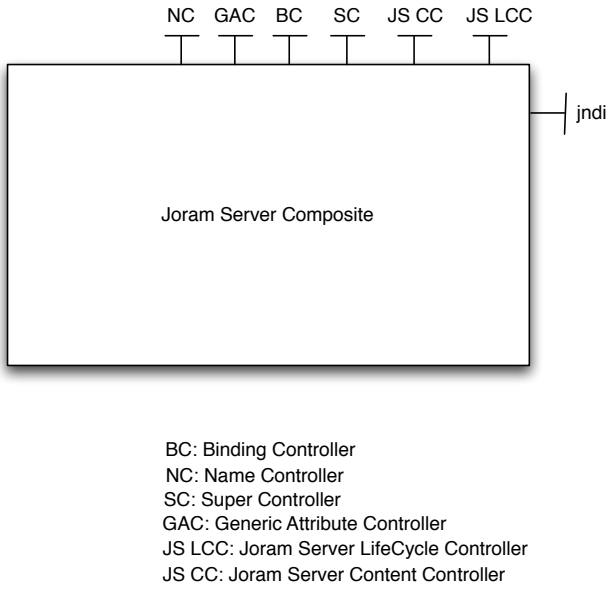

Figure 7: Fractal composite Joram server

classpath, an *a3servers.xml* file describing the static architecture of the servers we want to deploy. This file and its particular structure is mandatory to start a server. It's only when this file is well written that a server can know its configuration and can start. As we can't know, at the begining, which architecture will be deployed, the configuration file is dynamically updated during execution.

If we want to deploy more than one server, it becomes mandatory to create a domain. It is done by using a connection to the *AdminModule* of the server previously created, and by calling the *addDomain* method. This call updates the Joram architecture configuration with the domain description. Then, the specific life cycle controller updates the *a3servers.xml* file by overwritting it with the *AdminModule* configuration updated with the new domain.

It's nearly the same if we want to add a new server to the domain. We have to connect to the platform *AdminModule* and call the *addServer* method. This call doesn't start the new server, but updates the *AdminModule* configuration with the new server configuration. The new server is thus able to know its own and other servers configuration. Then, the specific life cycle controller updates the *a3servers.xml* file and the new server can start (at Joram level).

Two cases must be distinguished when creating a server: is the server is the first of the domain or is it is just a new server in the domain?. According to that, the server start is different. In the first case, we have to:

- Create the Selfware component representing the server.

- Create a well-known *a3servers.xml* file with the first server description updated with its own parameters. This file doesn't declare any domain.
- Start the Joram server.
- If the server is part of a domain, update the AdminModule configuration to create a domain.
- Update the a3servers.xml file with the *AdminModule* configuration.

If we just add a server to an existing domain containing other servers, we have to:

- Create the Selfware component representing the server.
- Find a existing server part of the domain that the new server want to join. Connect to its *AdminModule* and call the *addServer* method.
- Update the a3servers.xml file with the *AdminModule* configuration.
- Start the Joram server.

In the server life cycle controller, the distinction between the first server and the others is made by instrospection. When a server component is created, the Fractal architecture is introspected to find if there is already one server component. According to that, the component can specify its start actions.

So, if the server to create is the first server, the life cycle will follow these steps:

- Introspect the Fractal architecture to find other servers.
- Create a weel-known a3servers.xml file with the first server description updated with its own parameters. This file doesn't declare any domain.
- Start the Joram server.
- If a domain name attribute exists, connect to the *AdminModule* and call the *AddDomain* method.
- Update the a3servers.xml file using the *AdminModule*.

If the server to create is only one server to add to an already existing domain that already has servers, the life cycle will follow these steps:

- Introspect the Fractal architecture to find other servers.
- Connect to the *AdminModule* and call the *addServer* method to update the platform configuration.
- Update the a3servers.xml file using the *AdminModule*.
- Start the Joram server.

Note: According to Joram specification, we deploy only one unique Joram server per Java Virtual Machine. It is also impossible to start two servers in one Java process. That's why we advise to use one Jade node for each server to deploy.

4.3 Destinations and Users

Queues, topics and users are modelized as Fractal primitive components. They have a specific functional interface that reflects their business interface, allowing to manipulate them through the Joram API.

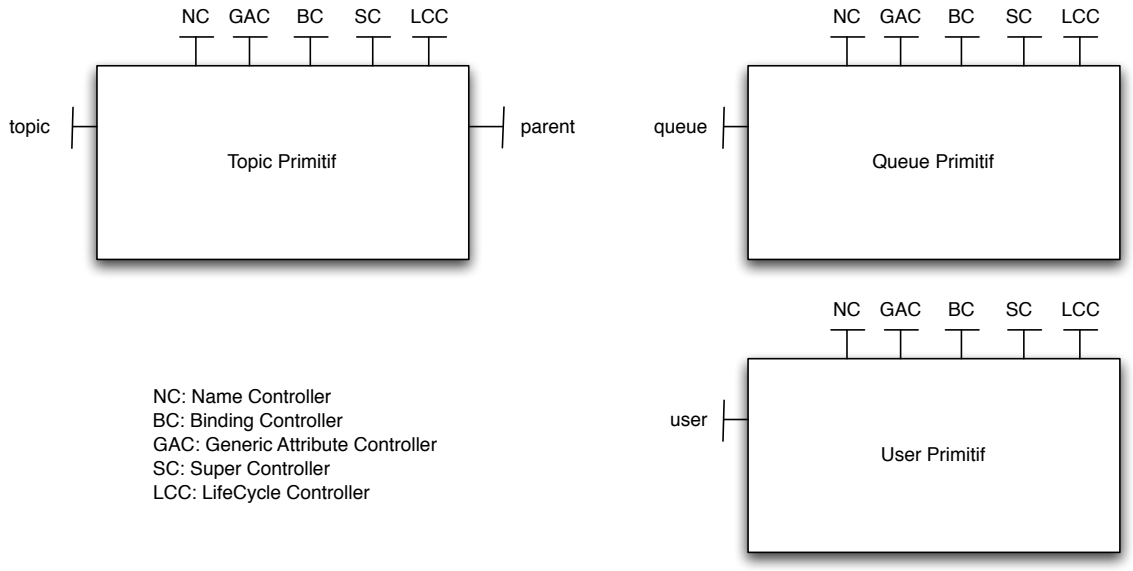

Figure 8: Fractal model for Joram elements

They are contained in the composite representing the Joram server. Moreover, topics and queues can be shared between servers and so, surrounded by the composite representing a server.

```
package org.ow2.jasmine.jade.resources.joram;

import org.objectweb.jasmine.jade.util.JadeException;
import org.objectweb.joram.client.jms.Queue;

public interface QueueInterface {
        public Queue getQueue();
        public void setProperties() throws JadeException;
        public String getMyServerSID();
}
```

```
package org.ow2.jasmine.jade.resources.joram;

import org.objectweb.jasmine.jade.util.JadeException;
import org.objectweb.joram.client.jms.Topic;

public interface TopicInterface {
        public Topic getTopic();
        public void setProperties() throws JadeException;
        public void unsetParent();
        public void setChildInHierarchy(String topicName, TopicInterface ti);
        public void unsetChildInHierarchy(String topicName);
}
```

```
package org.ow2.jasmine.jade.resources.joram;
```

```
import org.objectweb.joram.client.jms.admin.User;

public interface UserInterface {
        public User getUser();
}
```

4.4 Joram Connection Factory and Cluster Connection Factory

In addition to cluster queue, Joram introduces the concept of cluster connection factory. A cluster connection factory is composed of a set of connection factories. Like a cluster queue, a client who wants to create a connection, just needs to get the cluster connection factory. He is then route on a particular connection factory, thanks to optimization criteria. Cluster connection factories are represented in Selfware by composites Fractal components and connection factories by primitives one, shared by servers and cluster connection factories components.

5 Self-Repairing a Joram Application

Today JORAM is deployed in numerous operational environments where it may be used in several complementary ways. In the following, we report on an experiment of the self-repair function applied on a Joram application, where Joram is used as an independent messaging system between application sub-parts running in a Java environment on a large scale network. Section 5.1 describes the Joram Application, and its autonomic repair is presented in section 5.2. A quantitative evaluation of the self-repair function applied on this kind of applications is given in this section.

5.1 The Joram Application

The self-repair scenario applies on an Train Station Application used by a French public transport company. In its train stations, this company has several equipments installed, which have to be monitored and managed. Some main examples of these equipments are the following.

- Some mechanical devices such as escalators, fans.
- Some sound, telephony or video surveillance systems.
- Some display or printing devices.

These equipments can be distinguished not only by their main function but also by the protocol used to manage them: ModBus, SNMP, etc. Each equipment has a functional interface, independent of its protocol. This interface specifies the commands that can be triggered on the equipment and the several data that can be monitored on it: running state, alarm, etc. The aim of the Train Station Application is to remotely monitor and to manage these equipments in response to human orders. To do that, the application is divided in two parts, that both use the equipments functional interfaces:

- One, relocated, that provides equipments awareness to local staff. Its aim is to allow the Train Station staff to be aware of the state of operational readiness of equipments and to allow the staff members to manage equipments accordingly.
- The other, centralized, used by the maintenance service, that collects the state of all equipments.

The Train Station Application is also divided in two parts, as presented in the picture above. We can distinguish the server that monitors the equipments, from the client, connected to a server, providing the equipments awareness through a graphical interface. The architecture of this client/server application is presented in the figure 9.

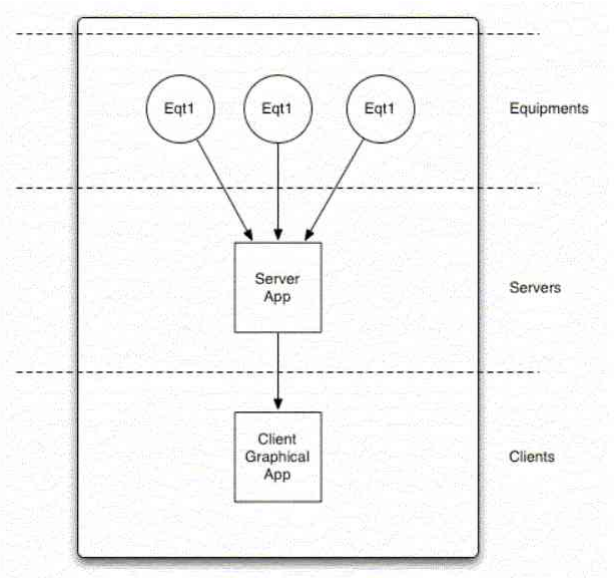

Figure 9: User worflow

Figure 10: Architecture of the Train Station Application

The implementation architecture of the Train Station Application is presented in the figure 9. In this architecture, the Business Object is an equipment specific object that allows manipulating the equipment through a JMS interface. It implements the equipment functional interface. This object is part of the server application. The proxy object is a client-side object that provides a remote access through JMS to the business object, and so, to the equipment functional interface. This object is invoked by the client graphical application.

The functional (i.e., business) interface of the Train Station Application defines two ways to manage the equipment. A synchronous way (Request / Response) that allows the proxy manipulating the business object, and an asynchronous way (Publish / Subscribe) that allows the business object to publish its running state.

The synchronous interactions are implemented through JMS queues on which, requests and responses are sent. The asynchronous interactions are implemented through a JMS

topic on which, proxy objects subscribe and business objects publish their running state and data.

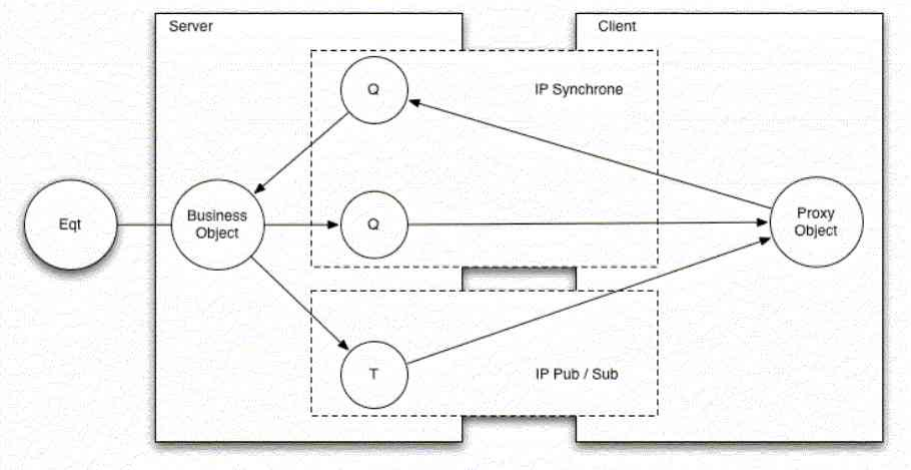

Figure 11: User worflow

Figure 12: JMS-based Architecture of the Train Station Application

5.2 Self-repairing the Joram Application

To be managed by the Selfware platform, the Train Station Application objects are encapsulated within Selfware manageable components. The queues and the topics used at the server side, as well as the Joram server, are encapsulated following the explanations given previously in this paper.

Once encapsulated, the global architecture of the application, expressed in terms of components, nodes and bindings, is given in order for the application to be automatically deployed by Selfware. Once deployed, the running application is automatically put under the control of the Selfware autonomic managers. Indeed, all the components described in the global architecture are monitored and automatically repaired in case of failures. The current version of the Selfware platform only manage hardware failures of nodes and *heisenbugs*[1], where failures detectors are based on heartbeat techniques. When one or more failures are detected, the self-repair manager determine the set of failed components through introspecting the global architecture to discover all the components that were deployed on the node that failed. As JMS based applications are loosely-coupled, they tolerate partial failures and allow the lost subsystems to be repaired and reinserted without requiring a global shutdown of the distributed system. In the present case, a failure impacting the queues or the topics does not imply the client part to be restarted.

In the following of this section, we evaluate our approach, discussing the illustrative JMS context in which Selfware can bring full autonomic self-repair. This evaluation has two purposes. On the one hand, it gives the opportunity to discuss concrete use cases of the Selfware technology, in particular discussing how difficult writing wrappers happens to

be. On the other hand, this evaluation allows us to discuss performance. Our goal with Selfware is not essentially performance-oriented as we aim to replace a human administrator whose efficiency to diagnose a failure and repair a complex distributed system is inherently low. Nevertheless, we felt that a raw evaluation was necessary for a complex middleware such as Selfware. The numbers confirm it, the ability of Selfware to repair a complex distributed system is almost instantaneous from a human perspective. From a machine perspective, we are in the ballpark of other standard recovery techniques such as the inet daemon, something in use everyday.

Considering the Train Station Application, once the Business objects mapped on equipments and the server application have been wrapped in Selfware components, the self-repair policy can be applied on them. The graphical Client application hasn't been wrapped because it is used at the client side (i.e., the client being a human Train Station administrator in this case) to be aware of the state of the equipments. In case of a failure occuring at the client side, the client will restart its graphical application.

Failures occuring at the server side may impact the business objects or the Joram servers of the Train Station Application. They normally require the intervention of a human, who has to detect and understand them. Assuming a software failure, he will reboot locally the failed business objects or server application. Assuming a hardware failure, he will have to setup another machine, configuring and starting these failed elements. Besides the error-prone process, the lower bound of the mean time to repair is dependent on the time necessary for a human to react and reconfigure the failed system.

With Selfware, the detection and recovery is automated. The self-repair manager automatically repair either business objects or the server application in case of a failure. It restarts the failed elements on an available host. The Mean-Time-To-Repair (MTTR) is more preciselly dominated by the time to detect the failure, the time to redeploy the necessary software on the newly allocated node, and finally to restart legacy systems.

To experiment this, we provoked failures on either the Joram server or the business objects. The self-repair manager detects and repairs these failures within 10 seconds. These numbers include the time for the failure detector to trigger and the time for downloading and installating the necessary software (Joram) on a new node. They include the installation of the Java wrappers and the apply of the overall management operations, including the writing of the configuration files from attributes. Ultimately, they also include the time it takes for Joram to initialize and start. While these numbers could be considered large, they are orders of magnitude better than any manual repair time, even by skilled operators.

From the clients of the Train Station Application (the administrators of the Train Station), the failures can be visible during the MTTR. They can see the occurrence of failures at the level of their graphical tool, that transcripts the state of the server application. As soon as the repair is processed by the self-repair manager, the tool returns to a normal state without having to be restarted.

Finally, we experimented with the self-self-repair behavior of the Self-Repair itself and its overhead on the ability of the Self-Repair to repair managed legacy systems. We forced a failure of one of the Self-Repair replica (including both a replica of the repair manager and the managed architecture as described in the SP1-L2 document). These failures are detected and handled in this experiment in one repair session. Hence, there is work to do for repairing the lost replicas of the repair manager and the managed architecture. The most important point is that the Self-Repair remains 100% available even facing partial failures of its replicas. A repair of a failed replica of either the managed architecture or the repair manager is done entirely in the background and introduces no disruption in the ability of selfware to repair any managed system.

6 Conclusion

We presented in this paper the autonomic repair capabilities of Selfware, an architecture-based management system for loosely-coupled distributed systems assembled from legacy systems. We believe that most distributed systems that are candidate to dynamic fault detection and repair are loosely-coupled, as are JMS legacy systems considered in this paper; they tolerate partial failures and allow the lost subsystems to be repaired and reinserted without requiring a global shutdown of the distributed system. In this context, Selfware provides a fully autonomic repair solution, a capability that we termed self-self repair.

Through the management of such loosely-coupled systems, our prototype has shown that the overhead of Selfware was negligeable. The achieved MTTR is largely dominated by the time required to detect a failure and the time to re-create and restart failed legacy systems. Selfware therefore delivers on its promise: remove the human administrator delays and potential errors out of repairing complex distributed systems. Using the very same management capabilities as a human administrator would use, Selfware delivers an MTTR that is very close to optimal.

References

[1] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 01(1):11–33, 2004.

[2] J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *IEEE Computer Magazine*, 36(1), 2003.

[3] Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. DREAM: a Component Framework for the Construction of Resource-Aware, Configurable MOMs. *IEEE Distributed Systems Online*, 6(9), September 2005.

[4] Sun Microsystems. JSR 914: JavaTM Message Service (JMS). http://java.sun.com/jms/.