Availability through Adaptation: a Distributed Application Experiment and Evaluation

Vania Marangozova, Daniel Hagimont
SIRAC Project (INRIA-INPG-UJF) INRIA Rhône-Alpes
ZIRST, 655 avenue de l'Europe, Montbonnot 38334 St Ismier cedex (France)
Vania.Marangozova@inria.fr, Daniel.Hagimont@inria.fr

Abstract

This paper presents our approach to distributed-application adaptation, focusing on the support of various execution contexts. We adapt an existing distributed component-structured application without modifying the application code. We describe our experiment in which we use replication-based adaptation in order to guarantee application availability in a faulty environment.

1 Introduction

In a world of increasing platform diversity, adaptation in distributed applications and systems is becoming crucial. The rapidly evolving market, especially in the mobile devices' case, demands new versions of already existing distributed software. Adaptation allows to react to the market evolution and to rapidly provide the same services under different execution contexts.

In this article we consider adaptations of existing component-based applications. We are interested in guaranteeing application availability under different network conditions. The adaptations we look for are to be done without application code modifications, thus favouring code-reuse and reducing software production costs. To meet these adaptation objectives, we use the JavaPod experimental platform implemented in the SIRAC project [1]. JavaPod provides mechanisms for the separation, the creation and the composition of different software aspects [6]. Since we focus on availability, the two particular aspects manipulated are replication and consistency.

This paper is structured as follows. Section 2 treats the adaptation problem and its projection in the availability case. Section 3 describes the adaptation scenario considered in our experiment. Section 4 details the JavaPod platform and our implementation of the scenario. An evaluation and some concluding remarks are proposed in Section 5.

2 Availability through adaptation

We consider the problem of adaptation in the context of distributed component-based applications. In this case, a distributed application is defined as a set of interconnected components. Components are graphs of objects providing interface-declared services. We allow the existence of compound components, even though most commonly used component models like COM [2] or CCM [8] follow a one-level hierarchy restriction.

We define the adaptation of a component-structured application as a process which:

- does not modify the code implementing the functionality of a component (its *functional properties*),
- does modify the application through component extension and reconfiguration.

We think that such an adaptation is greatly facilitated by aspect separation [8], i.e. separation between the functional properties and the rest of the application code (the *non functional properties*). Non functional properties in distributed applications include for example resource management, communication protocols, protection, persistency, etc.

Classic distributed software implementations do not provide separation of aspects. CORBA's [10] object services (non functional properties treatments), for example, need to be explicitly referenced by the programmer. A precedent in the aspect separation domain is the EJB specification [9] with its implicit management of transactions and persistency. EJB does not, however, allow to consider other non functional properties or to choose between different aspect managements.

In our work we focus on the question of high application availability under different network conditions (e.g. faults, intermittent connections, etc.). Our main interest is, therefore, the possibility of separation and adaptation of the replication and the consistency aspects according to these network conditions. Projects interested in component adaptations involving the aspects of replication and consistency are rare, despite the fact that current projects interested in aspect separation and configuration are numerous (e.g AspectJ [7] at the application level, FlexiNet [4] with its communication stack's configuration, etc.).

We propose an experimental study of the adaptations of the replication and consistency aspects. We consider a distributed application constructed to work in an "ideal" network environment (i.e. no faults). We adapt it by adding the aspects of replication and consistency and thus guarantee its availability in a faulty environment. The application, as well as the adaptation scenario, are described in detail in the next section.

3 Adaptation scenario for a distributed application

The application considered in this experiment is a distributed agenda used at the INRIA Rhône-Alpes institute for conference room reservations. The application has a standard client-server architecture. The client has a graphical interface which transforms all user actions into requests to the server. The server is responsible for all reservation manipulations (creations, editions, suppressions). There are no manipulation conflicts as there is only one server which serializes all incoming requests. The implementation is not fault-tolerant. In the case of host or network failures, the reservation application is not available.

The goal of our adaptation scenario is to make the reservation application tolerant to server faults. Fault-tolerance in this case means that the application is able to respond to client requests normally, even when a server is down. This implies that other servers are to be available and that clients are to be able to redirect themselves transparently. Transparent redirection is only possible if servers' states are kept consistent. Hence, to guarantee availability in this case, we need to manage the following additional aspects:

- Copy creation: It is important to decide what server components are to be replicated. If load balancing is considered, the solution of partial replication and function distribution will be acceptable. Since we wish that switching between servers be transparent for the clients, we need to replicate the totality of the server components.

- Group management: Since we have multiple replicas, we need to manage replica placement and group membership (server failure and removal, server creation and insertion). Group membership may be static (members are known a priori) or dynamic.

- Client-server connection: We need to manage connections in order to prevent clients from connecting to a faulty server as well as to redirect clients upon a server failure.

- Consistency management: To meet the goal of fault transparency for clients, we need a strong consistency management. All modification requests must be executed atomically on all correctly functioning servers.

4 Experiment and evaluation

This section presents in detail our application adaptation experiment. The programming language used is Java. We begin with a brief description of the JavaPod platform used throughout this implementation. Having presented the base JavaPod version of the reservation application and restated our adaptation objectives, we continue with programming details on the adaptation scenario. We conclude with an evaluation of the adaptation advantages of such a platform as well as with a feasibility analysis of adaptation in the replication-consistency case.

4.1 The JavaPod platform

The JavaPod platform is a middleware kernel for distributed component-based applications. Its main goal is the transparent configuration of the non functional properties of an application. The list of non functional properties is not predefined (as in the EJB) but is open and extensible. JavaPod is written entirely in Java.

4.1.1 Concepts

JavaPod defines three main concepts: server, container and connector.

The server concept comes from the EJB specification. A server provides execution support for containers which, in turn, provide execution support for components. Servers provide system services like communication protocols, resource management, database management, etc.

Containers (also EJB-inspired) represent the system part of components. Through component encapsulation and interposition they manage properties like persistency, synchronization, replication, etc. Containers use the server-provided services. For example, if a server provides a database, a container makes the binding between the database and a persistent component.

The connector, an ODP-derived concept [5], is specialized in communications between components. Connectors are an abstract notion represented in the platform by a set of stubs and skeletons implementing any type of binding (RPC, asynchronous messaging, multicasting, etc.).

4.1.2 Composition model

All four types of entities (servers, containers, stubs and skeletons) are implemented as *compound objects*. A compound object is composed of a totally ordered set of objects (sub-objects). The first sub-object in this order is called *extensible object* while the others are *extensions*. While the extensible object is invariant, any of its extensions can be added, removed or replaced dynamically. Extensions can modify the extensible object's behaviour by overriding existing methods and by defining new ones.

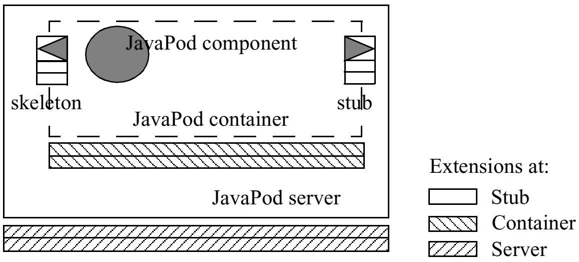

FIG.1 - *The JavaPod architecture*

Through the mechanism of extensions, servers, containers, stubs and skeletons are adaptable. As the basic forms of these components are completely generic, it is the extensions' configuration which defines the platform's non functional properties.

4.2 Base version of the application

We have reprogrammed the base version of our application so as to place ourselves in the JavaPod context. We have used JavaPod predefined classes for the client and the server implementations. In both (client and server) cases, the `Component` class is extended and only the functional code is specified. In the client, interface requests are transformed into calls to the server whereas the server implements its functional interface called `ServerAgenda_itf`.

As JavaPod does not allow the manipulation of compound components, we have implemented our server as one component rather than a graph of components. This solution defines the server as the unity of manipulation (replication, consistency, etc.).

As it can be seen on Figure 2, there is no explicit use of a particular communication protocol. Communication is actually added later, using the JavaPod extensions at the stub level. In our implementation we use a RMI-like communication. The two extensions involved are `Skeletonx` and `Stubx`. `Skeletonx` is added on the server (skeleton) side during the export of the server reference through the `bind` method. `Stubx` is added on the client (stub) side while importing the server reference through the `lookup` method. This mechanism is shown at Figure 3.

```
import javapod.*;
public class ClientAgenda
  extends Component {
  ServerAgenda_itf server;
  public ClientAgenda (ServerAgenda_itf
         server) {...}
  public boolean createEvent(Reservation
                             reservation){
    boolean result = true;
    try { result =
      server.createEvent(reservation);}
    catch (Exception e) {
      e.printStackTrace();}
    return result;}… }
```

```
import javapod.*;
public class ServerAgenda
  extends Component implements ServerAgenda_itf {
  private ResourceManager resourceManager;
  private ReservationManager reservationManager;

  public boolean createEvent( Reservation
                              reservation){
    boolean result = true;
    try {
    reservationManager.addReservation(reserva-
tion);
    } catch (OccupiedPeriodException e) {
      e.printStackTrace();
      result = false;}   return result;}… }
```

FIG.2 - *Functional code of the client and the server in the reservation application*

4.3 Adaptation approach

In our implementation, the true non functional properties are managed as extensions at the stub and possibly the container levels. This is done, for example, for communication and replica group managements.

As far as replication and consistency are concerned, these aspects involve component state management. State manipulation can be done either by violating the component encapsulation principle, or by adding replication-consistency specialized code at the component level. We choose the latter approach which allows the use of application specific information for the construction of efficient replication-consistency protocols. The specialized treatments are used by the means of upcalls.

```
public final void bind (Object o, String itf,
       String name) throws Exception {
  Container cont = ...
  ExtensionSet set = new ExtensionSet();
  set.add(new Skeletonx());
  Reference ref =
    cont.exportReference(o,itf,set);...}
```

```
public final Object lookup (String itf,
       String name) throws Exception {
  Reference ref = ...
  Container cont = ...
  ExtensionSet set = new ExtensionSet();
  set.add(new Stubx());
  return
    cont.importReference(ref,itf,set); ... }
```

FIG. 3 - *Communication configuration using the extensions Stubx and Skeletonx*

4.4 Adaptation for fault tolerance

We assume that servers are fail-stop.

To facilitate replication and consistency management we have modified the code and added state manipulation treatments (use of the “get” and “set” pattern).

The “get” and “set” methods are used by an extension called `ServerStateMgtExtension`. It provides two methods: `Capture` and `Restore`. The `Restore` method is used upon a dynamic server creation or after a server recovery. The newly arrived server replaces its state by the captured (`Capture` method) state of a correctly functioning server.

In order to prevent misfunctioning and inconsistencies during state capture, we have provided a `ServerLockExtension`. It is responsible for locking the state during the capture treatment. This solution is acceptable as we consider components without internal parallelism. If it is not the case, a more sophisticated treatment, ensuring that the component is in a stable state, must be used.

The replica group is managed by a `GroupMgtExtension`. Extending the container, it possesses the list of correct servers. This list is updated by a `ClientGroupMgtExtension` (at the client’s stub) and by a `ServerGroupMgtExtension` (at the server’s skeleton). The `ClientGroupMgtExtension` is responsible for the transparent redirection of the client to a correctly functioning server when a fault is detected. When the redirection succeeds and a new connection is established, the client and the server exchange their group views. The initial group view is given by an administrator.

A strong consistency management is guaranteed by a `TransactionMgtExtension` implementing the atomic execution of write requests. This extension does not change the client side invocation. In return, on the server side it is responsible for propagating the request to all the other correctly functioning group members. This is done using the information managed by the `GroupMgtExtension`. As we ignore network failures, a request propagation failure induces an update of the server's group view.

The difference between read and write requests, used by the `TransactionMgtExtension`, is done by a `RWExtension`.

The resulting architecture is shown at Figure 4.

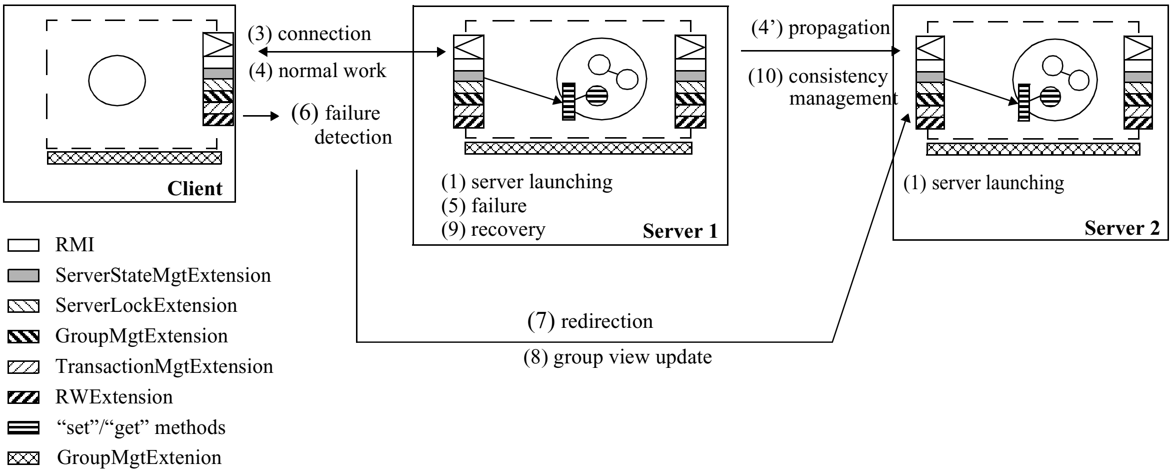

FIG.4 - *Adapted application architecture for fault tolerance*

4.5 Evaluation

Let us see in what extent our experimental results answer our adaptation objectives.

The first remark we would like to make is that adaptation is not possible if no suitable model is used. The adaptations we have realized with our reservation application have only been possible due to the JavaPod extension mechanisms.

Adaptation without component code modification is feasible with some exceptions. In fact, the truly non functional properties can be implemented separately from the application code. This is the case for example of communication management. In fact, the related treatments can be done through interception and encapsulation of inter-component interactions and do not need information on the components' internal structure. Their adaptation is thus done without component code modification. The case of the replication and consistency aspects is different. These aspects are closely related to the state and the semantics of the application components. It is therefore difficult to call them "non functional properties". Their integration in an existing application demands some component code modifications. Nevertheless, these modifications concern exclusively state capture and restoration. By using these state manipulation primitives it is possible to construct replication and consistency protocols in a modular way.

In our extensions' implementation as well as in our network conditions' treatment, we use no specific model. For the moment we do not have a mechanism allowing to analyze a specification of the network environment and of the desired replication-consistency management and to generate automatically the corresponding JavaPod extensions stack. Nevertheless, we believe that this can be done and that the adaptations can be organized according to a model taking into account both the component structure and the execution environment. As far as the component structure is concerned, the pattern approach [3] seems promising. Patterns give informations on application code and allow automatic code generation. Concerning the execution environment model there are already several interesting works treating application quality of service [10].

The JavaPod platform and its adaptation capacities have been only partially used. We have considered almost exclusively stub-level adaptations. We have appreciated the manipulation facility of the extension mechanism. Nevertheless the three-level (server, container, stub) adaptation seems difficult to use. It is not clear how the replication-consistency treatments are to be devised between these three levels.

We believe that a compound-component model would have facilitated and improved our organization of the replication and consistency aspects management.

5 Conclusion and perspectives

We have presented our experiment with adaptation of an existing distributed application. We have considered adaptations done through extension and reconfiguration rather than through code modification. We have described the implementation of an adaptation scenario focusing on application availability and more precisely on the case of fault tolerance. Most of the aspects added during adaptation can be treated separately from the application functional code. Unfortunately, this is not the case of replication and consistency which are closely related to the application semantics and the components' state. Nevertheless, we believe that further analysis of the possible organization and minimization of adaptations is worth working on. The open question of an adaptation model for the aspects of replication and consistency is to be investigated. Some clues for a solution allowing the consideration of the component structure and of the execution environment, may be found in works treating software patterns and network formalization.

References

1. E. Bruneton, M. Riveill. JavaPod : une plate-forme à composants adaptable et extensible, Rapport de recherche INRIA RA RR-3850. INRIA Rhône-Alpes, January 2000.

2. G. Eddon, H. Eddon. Inside Distributed COM. Microsoft Press 1998.

3. E. Gamma, R. Helm, R. Johnson and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

4. R. Hayton, A. Herbert, and D. Donaldson. Flexinet: a flexible, component oriented middleware system. SIGOPS'98, Sintra, Portugal, September 1998.

5. ITU-T Recommendation X.901 ISO/IEC 10746. Reference Model - Open Distributed Processing.1995.

6. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. ECOOP'97, Jyväskylä, Finland, June 1997.

7. C. Lopes, G. Kiczales, Recent Developments in AspectJ. ECOOP'98 AOP Workshop, Brussels, Belgium, July 1998.

8. Object Management Group. The Common Object Request Broker Architecture and Specification, Revision 2.4. OMG October 2000. 10

9. Sun Microsystems. Enterprise Java Beans Specification Version: 2.0. Sun Microsystems. October 2000. 11

10. R. Vanegas, J. Zinky, J. Loyall, D. Karr, R. Schantz, D. Bakken. QuO's Runtime Support for Quality of Service in Distributed Objects. Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), 15-18 September 1998 12