An Infrastructure for CORBA Component Replication*

Vania Marangozova and Daniel Hagimont

SARDES Project, INRIA Rhône-Alpes ZIRST,
655 av. de l'Europe, Montbonnot 38334 St Ismier cedex, France,
Vania.Marangozova@inria.fr,Daniel.Hagimont@inria.fr

Abstract. Traditionally applied to availability problems in various distributed computing domains (caching, fault-tolerance, disconnections), replication solutions remain difficult to implement and challenging to reuse. In this paper we propose a component-oriented approach in which replication is treated as a part of the configuration/reconfiguration aspect of an application. The approach allows an easy reuse of replication solutions and their integration through adaptation in existing distributed component-based services. We apply the approach to an implementation of the CORBA component model: OpenCCM.

1 Introduction

Traditionally applied to availability and performance problems in the areas of fault tolerance, caching and disconnection management, replication gains considerable importance with recent developments in global and mobile computing. However, the emerging highly-dynamic environments ask for new replication solutions. The problems of rapid and easy conception, reuse and adaptation of existing replication solutions are more topical than ever.

Due to their lack of generality, current replication solutions do not respond to the needs of reuse and adaptation. In fact, they apply to specific computing domains (databases, web management, etc.) and depend strongly on the underlying system architecture. As a result, even if identical replication principles apply to different contexts, replication solutions' reuse remains a real challenge.

Component-based architectures are a promising approach in the quest of a generic replication environment. In fact, replication solutions' domain-specificity limitation can be dealt with using the component encapsulation principle. Furthermore, the need of reuse and adaptation of replication solutions integrates well in the logic of the fundamental software(component) reuse principle.

In this article we investigate on an adequate infrastructure support for replication integration in component-based systems. We describe a deployment/reconfiguration approach to replication and show the facility with which different protocols can be attached to components without modifying their core business

* This research is partially funded by the RNTL project ARCAD.

J. Bishop (Ed.): CD 2002, LNCS 2370, pp. 222–232, 2002.

code. Our work is based on the OpenCCM [11] platform: an implementation of the CORBA Component Model (CCM) [12].

The article is organized as follows. Section 2 details replication issues in component-based systems. Section 3 describes the used OpenCCM platform. Our infrastructure for replication configuration, as well as its application to two replication scenarios, is presented in Section 4. Sections 5, 6 and 7 discuss respectively the lessons learned, related work and future perspectives.

2 Replication in Component Systems

Contrary to the object-based platforms centered on distributed application development, the component paradigm considers the entire application life cycle. Paying a particular attention to applications' administration, the paradigm promotes the separation between the components' business logic implementations and the system services they use. In component-based middleware like Enterprise Java Beans [14], this principle is reified by *container* servers hosting component instances and managing component-associated system services in a separate way. Components are thus reused, without modifications of their business code, in the context of different applications with different system management requirements.

A component-based replication infrastructure, aiming at component reuse in different replication contexts as well as at replication solution reuse in different application contexts, should provide adequate solutions to two major points. The first point concerns the choice and the mechanisms for creating and placing copies on different network nodes (*replication*) while the second considers the relations established between these copies (*consistency*).

Replication configuration should be a main characteristic of a flexible replication management solution. It should be possible to configure the set of replicable entities (*what*), to define the most appropriate moment for replication (*when*) and to control optimal copy placement (*where*). This is a rather ambitious objective given that standard distributed systems use either no replication (remote procedure calls) or fixed replication schemes with predefined and systematically used replicated entities. Java RMI [15] or the new CORBA3 standard do allow the co-existence of replicable and remotely accessible objects but do not allow the switch between the two without functional code modification. A *non intrusive* replication management in a component-based system should prevent functional modifications and allow for a future integration of the corresponding treatments in a container. Replication will thus be naturally considered as a part of the component's configuration and administration.

Consistency configuration and adaptation is a major objective for a target infrastructure intended to allow reuse of components with different replication scenarios. This conclusion has emerged after a considerable research on consistency showing the inexistence of a universal protocol [2], the insufficiency of application specific solutions [3] and the encouraging results on consistency adaptation [13]. Consistency should be therefore managed in the same way as

replication: it should be part of the component's configuration and administration.

Before describing the principles of our infrastructure for flexible component replication, we present the platform used for our work: OpenCCM.

3 The OpenCCM Platform

In order to address deployment and administration issues in distributed applications' life cycles, the CORBA standard proposes the CORBA Component Model (CCM) [12]. The model defines a server side component framework integrating component interface specification, component implementation, application deployment and execution. Implemented at the University of Lille I, OpenCCM [11] is an available open source, Java-based, CORBA-compliant, partial implementation of CCM.

3.1 OpenCCM Components

OpenCCM component types are explicitly declared in terms of components' used and provided interfaces (*ports*). Fig.1 gives the IDL declaration for an agenda application in which users can connect to an agenda server and register, edit or remove rendezvous from their plannings. The corresponding `ManageReservations` interface is *provided* by the `Server` component and *used* by the `Client` component.

OpenCCM component instances, as well as their corresponding ports, are represented by standard CORBA objects that we call respectively component objects and port objects. A component object references the corresponding component implementation and all component's ports objects. This structure is the basis of the OpenCCM introspection facilities. During deployment, introspection is used to acquire port references and to establish component interconnections. At runtime, introspection allows to explore these interconnections and to access the corresponding port references needed for component method invocations.

```
struct Reservation{...};      typedef sequence<Reservation> listReservations;

interface ManageReservations {                          //Business interface
    void addReservation(in Reservation res);
    void RemoveReservation(in string resId);
    listReservations getReservations();

component Client {                                      //The client component
    uses ManageReservations to_S;}                      //Used interface

component Server {                                      //The server component
    attribute name;                                     //Configuration attribute
    provides ManageReservations for_C;}                 //Provided interface
```

Fig. 1. A component description of a simple application

3.2 OpenCCM Containers

OpenCCM does not implement a separate container entity but integrates container functions in component implementations using specialization of the component generation process. The component inheritance tree is actually enriched in order to include specific OpenCCM classes defining introspection and port interconnection operations. The resulting component implements a standard CORBA IDL interface (obtained through a mapping from the initial description) containing both the business interfaces and the additional introspection and port management interfaces. In the case of the agenda example, the mapping produces the `Client` and `Server` interfaces shown in Fig.2.

The predefined `CCMObject` interface is responsible for providing all generic introspection operations. The `Client`'s `get_connection_to_server` and the `Server`'s `provide_for_clients` methods are introspection operations returning the references to the respective `ManageReservations` ports (corresponding to the used and provided interfaces). A `Client` instance uses the obtained reference to invoke a `Server` component. The other operations are used for connection management and are discussed in the next section.

```
interface Client : CCMObject { ...Fig.1. business code      //Component Client
    void connect_to_S(in ManageReservations cnctn);         //Port management
    ManageReservations disconnect_to_S();                   //Port management
    ManageReservations get_connection_to_S();}              //Introspection

interface Server : CCMObject {...Fig.1. business code       //Component Server
    ManageReservations provide_for_C(); }                   //Introspection
```

Fig. 2. Component-oriented standard IDL description of the agenda

3.3 OpenCCM Deployment

OpenCCM deployment is done by a deployment program which includes statements for component archives installation, for component instance creation, for component configuration and interconnection, and for application launching. Component interconnection is done using the port management interface. In the case of our agenda application, the `Client`'s port management interface contains the `connect_to_server` and `disconnect_to_server` operations through which the client is connected and disconnected to the server (Fig.2).

OpenCCM defines a basic deployment environment manipulated through a simple API. The way it is used in a schematic deployment program for our agenda application is given in Fig.3. The basic steps include the choice of the deployment hosts (1), the installation of components' implementations (2), the creation of component instances (3), their configuration (4), the component interconnection (5) and finally, the application launching (6).

```
//(1) Obtain the deployment servers
// ns is the CORBA's NamingService
ComponnentServer cs1 = ns.resolve("CS1");

//(2) Install component archives
//inst is the cs1's installation factory
inst.install("agenda","agenda.jar");

//(3) Create components
//sh/ch are components' instance managers
Server s = sh.create();
Client c1 = ch.create();

//(4) Configure components
// s.name("Server");

//(5) Connect client and server
// Get ports using introspection
ManageReservations for_C=
        s.provide_for_C();

// Establish connection
c1.connect_to_S(for_C);

//(6) Launching
s.configuration_complete();
```

Fig. 3. OpenCCM deployment of the agenda

4 Replication Management in OpenCCM

We discuss the design choices for an adaptable replication management infrastructure in the first part of this section. In the second part we describe our experience in which we apply the proposed principles to two replication scenarios for the agenda application.

4.1 Principle

Non intrusive replication management (cf. section 2) requires mechanisms configuration of both the replication and consistency aspects.

We integrate **replication configuration** at the deployment level (OpenCCM's deployment programs). In fact, copy creation and placement spell well in terms of application architecture configuration which is itself defined during deployment. CCM deployment actually describes *what* and *where* components are to be deployed, how these components are to be connected and is meant (even if it is not the case yet) to specify architectural reconfigurations (*when*). The replication aspect adds the specification of the set of replicable components (*what*), the best copy placement (*where*), the consistency management (*how*) and the most appropriate moment for replication (*when*).

We base **consistency configuration** on interception objects, consistency links and component-specific state management. *Interception objects* allow to integrate consistency management without modification of the components' business code. They intercept copy invocations and trigger consistency actions. The latter take the form of pre and post treatments for the intercepted method calls which continue to be delegated to the initial component implementations.

Consistency protocols define consistency relations between copies and provide treatments to maintain these relations valid. Logically, these treatments require the existence of copy interconnections to propagate consistency actions. We call these connections *consistency links*.

Most consistency protocols access component internal data. In our prototype we preserve the component encapsulation principle but take advantage of a component-specific state management by leaving the state access primitives

implementation to the component developer. Consistency protocols can thus use these primitives and ignore component implementation details.

The concrete interception objects' and consistency links' implementations depend on the specific consistency protocol chosen for a given application. The next section describes the way these entities are generated in order to provide replication-aware deployment in the OpenCCM platform.

4.2 Implementation

As discussed in the previous section, our component replication management is based on interception objects used to catch component invocations and to execute consistency treatments, on consistency links used to interconnect replicas and on component accessor functions used to manipulate internal data.

An **interception object** in our prototype is a component object implementing the same interfaces as the corresponding component but whose code contains the consistency protocol implementation. The functional code of a component is managed in a separate object and is referenced by the interception object for invocation propagation. Notice that this design is equivalent to the interposition object used to manage containers in the EJB component model.

The **consistency link implementation** requires that a consistency protocol expert (who is also responsible of the implementation of the consistency actions) define the nature of the interfaces between the component copies. The interfaces have to be IDL-described and are used to generate the final replicable component.

As mentioned in the previous section, the **component state manipulation** treatments are provided by a developer who is aware of the component's semantics. For Java components, we provide a default implementation based on Java Serialization.

Generation of replication management code i.e. the definition of the consistency link interfaces, the integration of the consistency protocol implementations and finally the adaptations to the applications deployment programs is currently done by hand. Tools for automation of the pretty systematic replicable component generation process are under development.

In Fig.4, both the component to be replicated and the consistency protocol to be applied are represented by their IDL definitions and their implementations. The consistency protocol's definition declares the consistency links interfaces involved in the component copies' coordination. The protocol implements these interfaces as well as the component's business interfaces in order to intercept the corresponding invocations. The resulting replicable component implements the interfaces and includes the implementations of both the initial component and the chosen consistency protocol.

The procedure for integrating a replication scenario in a given component-based application involves the following actions. First, a developer is to provide the **component business-logic implementations**. If a component is to be replicated, he will need to provide primitives for state capture and restoration. In order to integrate a **consistency protocol implementation**, a replication

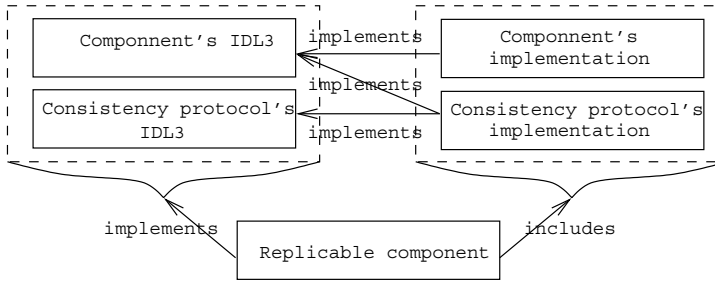

Fig. 4.

expert has to implement the corresponding protocol and to provide its IDL interfaces. The **generation of the replicable components** is based on the former actions: the component and consistency IDL definitions are merged and associated with the corresponding implementations to generate the resulting replicable component. Finally, the **replicated architecture deployment** may take place. Components and their copies, as well as component interconnections including the needed consistency links are explicitly created in the deployment program.

4.3 Experience

We have applied the above infrastructure principle to two replication scenarios for the agenda application. The first one implements a simple disconnection scenario while the second implements a caching system.

Disconnection protocol. In the agenda's disconnection scenario we keep the same component business implementations (The IDL descriptions are those of Fig.1 and Fig.3). We just make the `Server` component `Serializable` and benefit from a default state management procedure including the two primitives `State captureState()` and `void restoreState(State state)` granting access to the component's internal data (of type `State`).

The consistency protocol (Fig.5 shows the IDL definition) distinguishes between master and slave server copies (`role` attribute). A slave server is a disconnected copy of a master server. When a disconnection process is launched, the slave server is created on the machine getting disconnected and initialized with

```
component Server {
   attribute Role role;
   provides DiscPtcl for_disc;
   uses DiscPtcl to_disc; }
```

```
interface DiscPtcl {
   void make_copy();
   void reconcile();
   State get_state();
   void push_log(in Log log); }
```

Fig. 5.

```
public ServerImpl() {
  if (role.isMaster())
     realObj = new ServerActualImpl();
  else log = new SimpleLog(); }

public void
addReservation(Reservation res) {
  realObj.addReservation(res);
  if (role.isSlave()) log.put(res);}
```

```
public void reconcile() {
  to_disc.push_log(log)};

public State get_state() {
  return realObj.captureState();}

public void push_log(Log log) {
  realObj.restoreState(state);}

public void make_copy() {
  realObj = to_disc.get_state();}
```

Fig. 6.

```
//(1) Create components
srv = SFactHost2.create();
srv.role("Master");
clnt = CFactHost1.create();

//(2) Connect client and server
ManageReservations for_C =
srv.provide_for_C();
clnt.connect_to_S(for_C);

//(3) DISCONNECTION
//Create a copy
copy = SFactHost1.create();
copy.role("Slave");
```

```
//(4) Update copy before disconnection
copy.makeCopy();

//(5) Update connections
clnt.disconnect_to_S();
for_C= copy.provide_for_C();
clnt.connect_to_S(for_C);
DiscPrtcl = srv.provide_for_disc();
copy.connect_to_disc(for_disc);

//(6) RECONNECTION
copy.reconcile();
clnt.disconnect_to_S();
clnt.connect_to_S(srv.provide_for_C());
```

Fig. 7.

the state of the master. At reconnection, the possibly diverged slave and master states are reconciled. Reconciliation is based on a simple redo protocol using a log of disconnected operations.

The consistency protocol implementation is shown in Fig.6. The interception object holds a reference to the component's actual implementation and forwards invocations e.g. `addReservation`. If the current component is a slave copy, the interception treatment logs the operation for further reconciliation.

Fig.7 shows the deployment program for the disconnectable application architecture. After creation of the server and client components (1), they are interconnected (2). Upon disconnection (3), a slave server copy is created. Its internal state is synchronized (4) and the connections are updated (5) in order to connect the client to the created copy. Upon reconnection (6), the two copies of the agenda are reconciled and the client is reconnected to the master server.

Caching protocol. For the agenda's caching scenario we have implemented a version of the entry consistency protocol (multiple-readers/single-writer). Inspired by the Javanaise system [6], the protocol associates a locking policy (read/write) to each method of the agenda server and ensures the consistency of cached copies before forwarding an invocation.

The sites where caching should be applied are specified in the deployment program. A master site stores the persistent version of the server component and a client may address either the remote master copy or a local replica. The

```
//consistency link server to client
interface Server2Client {
   State reduce_lock();
   void invalidate_reader();
   void State invalidate_writer();}
```

```
//consistency link client to server
interface Client2Server {
   State lock_read();
   State lock_write();
   void State reduce_lock();
   void augment_lock();}
```

Fig. 8.

consistency links between replicas (Fig.8) use operations for fetching (in read or write mode) an up-to-date component copy and for copy invalidation.

5 Lessons Learned

We have shown that it is possible to manage replication as an adaptable non functional property in a component-based system. Furthermore, we have identified deployment configuration and component interface implementation as the two conceptual places for non functional replication integration. Deployment is used for replication scheme definition in applications' architecture configuration. Interface interception is used for consistency management and is to be part of an adaptable container architecture (a major EJB as well as CCM research issue).

We believe that, in addition to the explicit deployment phase, introspection and port management are OpenCCM's most interesting features for non functional replication management. The explicit reference manipulation helps reference integrity preservation upon reconfiguration. At the moment of writing, EJB does not provide a similar functionality.

6 Related Work

Configurable replication is a major issue in works on distributed shared memory providing multiple consistency protocols [2] or in mobile databases projects introducing optimistic consistency and application-specific reconciliation[5]. CORBA-centered research is also very present with works on flexible caching. The CASCADE [4] project for example is based on the CORBA interceptor mechanism while Flex [10] uses object subclassing and object-personalized state capture. The cited projects do certainly consider replication configuration and in some solutions are rather close to the mechanisms used in our experiment. However, they remain domain-specific while our work aims at overcoming this limitation and reconciles issues coming from different domains.

Adaptation in general is a major objective in language platforms interested in easy source code modification [9], in frameworks considering middleware architectures [1,7] and in operating system reconfiguration projects [8]. However, the existing works do not address the most difficult issue in replication adaptation: the analysis of the way replication can be defined as a separate aspect and the appropriate application of adaptation mechanisms.

Adaptation in component-based middleware is a very recent issue and existing adaptation efforts typically focus on non functional management of well specified properties like transactions, security and persistence but not replication. The presented work is one of the rare efforts on replication adaptation. In fact, most works treating replication in a component-based environment are EJB-oriented and apply fixed replication solutions based on replication of the underlying relational databases. In our knowledge, this is the first experiment with (non-functional) replication adaptation in the CORBA component model.

7 Conclusion and Future Work

We have investigated the integration of replication management in a component-based platform. We have proposed and implemented an infrastructure allowing to configure replication aspects in a non functional way. We have successfully integrated the proposed principles in the first Java-based implementation of the CORBA component model, OpenCCM, by preserving its CORBA compliancy. Our design is based on interception objects (to be comprised in the future OpenCCM's adaptable containers) and on deployment configuration extended to include replication policy description. We have defined the procedures for using this infrastructure and have shown its application in two scenario cases: a caching entry consistency system and a simple disconnection management.

An immediate perspective of this work is to provide the tools for automatic generation of the replicable components. Even if we have described the procedure, most of the replication integration work is done manually and can be automated. A tool for deployment program transformation (in order to integrate a specified replication policy) could also be provided.

Another interesting perspective of this work is the investigation of the way this infrastructure can be applied to other component models. Candidates are notably Microsoft's COM and EJB but also more abstract models like the standard ODP.

Finally, replication is only one system aspect and CCM one particular component model. The work presented in this paper takes part in a broader project aiming at the implementation of a generic and reflexive component-based middleware allowing encapsulation of different component types (EJB, CCM ...) and adaptation for transparent integration of non-functional system aspects (persistence, security, replication ...).

References

1. G. Blair, G. Coulson, P. Robin et M. Papathomas, An architecture for next generation middleware. In Proc. of Middleware'98. 191-206, Sept. 1998.
2. J.Carter. Design of the Munin Distributed Shared Memory System. Journal of Parallel and Distributed Computing, 29(2):219-27, 1995.
3. P. Dechamboux, D. Hagimont, J. Mossiere, and X. R. de Pina. The Arias Distributed Shared Memory: an Overview. In 23rd Intl Winter School on Current Trends in Theory and Practice of Informatics, LNCS 1175, 1996

4. G. Chockler, D. Dolev, R. Friedman, and R. Vitenberg. Implementing a Caching Service for Distributed CORBA Objects. In Proc. of Middleware'00,1-23, April 2000.
5. A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and B. Welch. The Bayou Architecture: Support for Data Sharing Among Mobile Users. In Proc. of the IEEE Workshop on Mobile Computing Systems and Applications, 2-7, Dec. 1994.
6. D. Hagimont, F. Boyer. A Configurable RMI Mechanism for Sharing Distributed Java Objects. IEEE Internet Computing, 5(1): 36-44, Jan.-Feb. 2001
7. R. Hayton, A. Herbert, et D. Donaldson. Flexinet: a flexible, component oriented middleware System. SIGOPS'98, Portugal, Sept. 1998
8. J. Helander and A. Forin, MMLite: A Highly Componentized System Architecture. Eight ACM SIGOPS European Workshop, Portugal, Sept. 1998.
9. G. Kiczales, E. Hilsdale, J. Hugonin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In J. L. Knudsen, editor, ECOOP 2001, Object-Oriented Programming, LNCS 2072. Springer-Verlag, June 2001.
10. R. Kordale and M. Ahamad. Object caching in a CORBA compliant system. USENIX Computing Systems, 9(4):377-404, Fall 1996.
11. R. Marvie, P. Merle, J-M. Geib, M. Vadet, OpenCCM : une plate-forme ouverte pour composants CORBA, CFSE'2, France, April 2001.
12. Object Management Group. CORBA Components: Joint Revised Submission. Aug. 1999. OMG TC Document ptc/01-10-26 (Components FTF interim report)
13. M. van Steen, P. Homburg, and A.S. Tanenbaum. Globe: A Wide-Area Distributed System. IEEE Concurrency, Jan.-March, 1999
14. Sun Microsystems. Enterprise Java Beans Specification Version: 2.0. 2000.
15. Sun Microsystems. Java Remote Method Invocation Specification. 1998.