An Infrastructure for CORBA Component Replication

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

Component-based programming is a promising approach to distributed application development. It encourages software reuse and promotes the separation (as in aspect-oriented programming) between the components' business implementations and the code managing the used system services. One system service, of particular importance to the distributed computing domain, is replication. It is actually used for various purposes e.g performance (caching), fault-tolerance, availability for mobile users. However, replication management in a component-based environment is still an open issue. This paper presents our approach to component replication and describes an experiment based on the OpenCCM platform (an implementation of the CORBA Component Model). Our approach has been validated with several applications.

1 Introduction

Replication is a classical solution to availability and performance problems in distributed systems. It is used to guarantee fault tolerance of services [20], to provide fast data access through caching [14], to guarantee service availability for mobile and frequently disconnecting entities [6], etc.

Despite their numerous applications to distributed computing, replication techniques remain difficult to implement. Actually replication is applied to distributed file systems, distributed databases, web document management, etc. but the provided solutions remain domain specific: they strongly depend on the manipulated entities' nature and on the distributed system architecture. As a result, the existing solutions lack generality and do not answer the question of what a generic replication service should be like.

Component-based architectures are a promising approach in the quest of a generic environment for distributed application development and execution. In fact, the *component encapsulation principle* is a good way to reconcile disparities among entities coming from different computing domains. Moreover, the fundamental *software reuse principle* motivates component reuse in different application and execution contexts. In consequence, providing a replication service in a component-based environment is a way to provide a generic replication solution which would apply to a wide range of application. Furthermore, since this replication solution should be used in many different contexts, it will have to allow different replication (and also consistency) protocols to be associated with components according to their contexts of (re)use. The issue is therefore to provide an adequate infrastructure support facilitating replication and consistency protocol integration in component-based systems.

In this article we report our experiment in providing an adequate infrastructure support facilitating replication and consistency protocol integration in component-based systems. Our

work is based on the OpenCCM [15] platform, a Java-based implementation of the CORBA Component Model (CCM) [16]. The principles of the proposed infrastructure are illustrated through two different scenarios for replication management. The first implements a caching system based on the entry consistency protocol while the second considers a simple disconnection management case. We describe our deployment/reconfiguration approach to replication and show the facility with which different replication and consistency protocols are attached to components without modifying their core business-logic code.

The article is organized as follows. Section 2 motivates research on replication in component-based systems. Section 3 describes the CCM model and the OpenCCM implementation that we use for our experiments. Our infrastructure for replication and consistency configuration, as well as its application to the two considered scenarios, is presented in Section 4. Sections 5 and 6 discuss respectively the lessons learned, the related work, and future perspectives.

2 Replication in component systems

Most distributed applications are built on top of *object-based* middleware platforms which provide services to encapsulate the distribution complexity. These platforms include the Common Object Request Broker Architecture (CORBA) [17], the Distributed Component Object Model (DCOM) [7] and the Java Remote Method Invocation (RMI) [23]. One major characteristic of these platforms is the use of Interface Definition Languages (IDL) (allowing compile-time type checking) together with automatic code generation for distributed communication.

However, if these platforms address the issue of distributed application development, they do not consider the entire application life cycle. In fact, software engineering requirements are better matched by the *component* paradigm which pays a particular attention to applications' administration. Aiming at isolation of the administration aspect, the paradigm promotes the separation between the components' business logic implementations (the *functional code*) and the system services they use (the *non functional code*). For this reason, component-based middleware like Enterprise Java Beans [22] define *container* servers to host component instances and to separately manage component-associated system services. Contained components are thus reused, without modifications of their business code, in the context of different applications with different system management requirements.

In this paper we investigate the integration of a *replication service* in a component-based infrastructure. We explore a replication solution allowing the implementation of various replication scenarios for a given component-based application. We are interested in a solution allowing the consideration of scenarios coming from the domains of cache management, fault tolerance and mobile disconnection.

Replication management includes two major points: replication and consistency management. Replication involves the choice and the mechanisms for creating and placing copies on different network nodes. Consistency is concerned with the relations established between these copies. In consequence, in order to support various replication scenarios, the target infrastructure should provide adequate solutions to the following issues:

- Replication configuration (*What, when & where?*)

Standard replication solutions impose fixed replication schemes with predefined and systematically used replicated entities [26]. These approaches are meant to guarantee performance and availability in cases where the Remote Procedure Call (RPC) systems [1] fail to

do so due to variable communication delays or network failures. Between the extremes of "systematic replication" and "no replication", there is a need for an intermediary approach allowing to decide *what* entities should be replicable. Approaches like Java RMI [23] or the new CORBA3 standard [16] allow the co-existence of replicable and remotely accessible objects. However, the choice between these two possibilities is static and is reflected explicitly in the components' application code. A flexible solution for replication management in a component-based approach should allow components to be replicable or remotely accessible without functional code modification. Moreover, such a solution should allow to decide of the most appropriate moment for replication (*when*) and to control optimal copy placement (*where*). The needed infrastructure should therefore manage replication in a *non intrusive* way. Such a non functional replication management will 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 (*How*?)

The distributed consistency issue has been mainly studied in the scope of distributed shared memory (DSM) systems. After the conclusion of the inexistence of a universal consistency protocol [3], DSM consistency research has focused on application specific solutions [5]. Yet, application specific consistency is not sufficient. There is a need for mechanisms enabling consistency configuration and adaptation depending on the context of replication use (DSM caching, availability during disconnection, etc.). Furthermore, recent results on consistency adaptation [21, 17] encourage investigations in this direction. Consistency management adaptation is therefore a major objective for a target infrastructure intended to allow reuse of components with different replication scenarios. This implies that, in the same way as replication, consistency should be managed in a non functional way, i.e. should be part of the component's configuration and administration.

In the next sections, we first describe the OpenCCM component-based platform that we use for our experimentations and then present the principles of our infrastructure for flexible component replication.

3 The OpenCCM platform

In order to address deployment and administration issues in the distributed applications' life cycle, the OMG CORBA [17] standard proposes a new model: the CORBA Component Model (CCM) [16]. Implemented at the University of Lille I, OpenCCM [15] is an available open source, Java-based, partial implementation of the CCM model. OpenCCM is the basis of the experiments described in this paper.

3.1 CCM: a model for component-based middleware

The CORBA Component Model is a new solution for component-based middleware. Many CCM notions are inspired by the previously defined Enterprise Java Beans (EJB) specification [22]. Like EJB, CCM defines a server side component framework in which components execute in *container*s, allowing the separation between business logic and system management aspects.

The CORBA Component Model defines several models corresponding to different phases in the applications' life cycle. In addition to the interface specification of components (*abstract model*), these models treat component implementation (*programming model*), deployment (*packaging and deployment models*) and execution (*execution model*). The CCM specification

mostly details the abstract model. Many issues in the other models remain open and need further investigations.

The abstract model is concerned with component descriptions. The defined descriptions are richer than standard IDL declarations as they consider both the interfaces used and provided by a component. This is made possible thanks to an IDL extension which introduces the notion of *port*. Ports define connection points (used and provided) for a component. Ports have a type which is an interface. They are used at runtime by clients for business method invocations and during deployment for component connections configuration. Given that the model imposes the static definition of all component's ports, the deployment phase makes all initial interconnections explicit.

CCM distinguishes between synchronous and asynchronous communication. Provided synchronous ports are called *facets*, while provided asynchronous ports are called *event sources*. The used ports are respectively *receptacles* and *event sinks*.

```
        struct Reservation {…} ;
        typedef sequence<Reservation> listReservations;
        interface ManageReservations {                          // Business interface
                void addReservation(in Reservation reservation);
                void removeReservation(in string reservationId);
                listReservations getReservations();
        };
        valuetype SeminarEvt … {                                //Event definition
                public string seminarDescription;
        };
        component Client {…                                     // The client component type
                attribute string name;                          // Attribute for configuration
                uses ManageReservations to_server;              // Used interface
                consumes SeminarEvt seminar;                    // Used interface
        };
        component Server{ …                                     // The server component type
                attribute string name;                          // Attribute for configuration
                provides ManageReservations for_clients;        // Provided interface
                publishes SeminarEvt seminar;                   // Provided interface
};  …                                                           // Other definitions
```

Figure 1 *An IDL3 description of a simple application*

Figure 1 gives an example of the IDL description of a simple application in this model. This toy application is an agenda management example which will be used throughout this paper. In this application, users can connect to an agenda server and register, edit or remove rendezvous from their planning. The server can send announcements of rendezvous (like seminars) to interested users. In the description we have the definitions of two components: `Client` and `Server`. The server provides the `ManageReservations` synchronous interface for rendezvous management. It is a receptacle (used interface) for the `Client` and facet for the `Server`. The seminar management is declared through the asynchronous `SeminarEvt` interface. The server publishes `SeminarEvt` events and subscribed clients consume them.

The programming model is meant to define the relation between a component and its container. As containers are intended to integrate non functional properties' management code (for replication management for example) generated from a specification of the component's needs, this model defines the nature of these generation rules.

The deployment model defines the deployment process which involves component implementations' installation (archives defined by the packaging model), component instances' creation and interconnection and application launching.

More details about the CCM models can be found in the OMG specification [16].

3.2 The OpenCCM implementation

OpenCCM [15] is a Java-based implementation of the CCM model. It provides an implementation of the abstract and deployment models, and a partial implementation of the programming model. OpenCCM is CORBA-compliant i.e. it uses the CORBA standard (CORBA2) as a basis and is implemented as an additional layer defining CORBA component (CORBA3) features. The platform is currently running on several existing CORBA implementations: ORBacus 4 for Java [18], OpenORB [19] and Visibroker 4 for Java [11].

3.2.1 Abstract model implementation in OpenCCM

The basic notions in the CCM abstract model are those of component and port. In OpenCCM, components, as well as component ports, are represented by standard CORBA objects (we will call them respectively *component objects* and *port objects)*. References to these objects are therefore standard CORBA references (see Figure 2).

A reference to a component object allows manipulation of the associated component e.g. for introspection purposes. The component object includes a reference to the component's implementation as well as references to all component's ports objects. This structure is the basis of the OpenCCM introspection facilities used mainly during the deployment phase. A deployment program (examples will be given further in this paper) uses introspection in order to acquire port references and to establish component interconnections. Port references are also used by clients at runtime for component business methods invocations corresponding to the declared used interfaces.

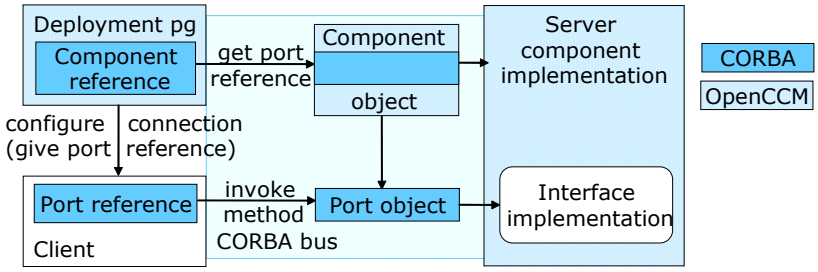

Figure 2 *OpenCCM platform architecture*

3.2.2 Programming model implementation in OpenCCM

The OpenCCM component implementation follows the inheritance-based CORBA model i.e. the implementation inherits from the CORBA object (skeleton). A component implementation consists of a unique Java class implementing all of the component's provided interfaces. In consequence, business method invocations made using port references are routed to and executed by an instance of this Java class.

OpenCCM does not implement the notion of container which is essential in the CCM programming model but allows specialization of the component generation process. This specialization, based on component object extension through inheritance, is used to associate policies for non functional properties management with components. The mechanism is used by OpenCCM itself in order to implement component port management and introspection. The

component object's inheritance tree is actually enriched in order to include specific OpenCCM classes defining introspection and port interconnection operations. Figure 3 shows the resulting structure for the Server component in our agenda application. The component's implementation (`ServerImpl`) inherits from `ServerCCM` which combines CCM features (introspection and port management defined in the predefined `CCMObjectOperations` interface) and CORBA standard management (the `Servant` class is ORB- provided).

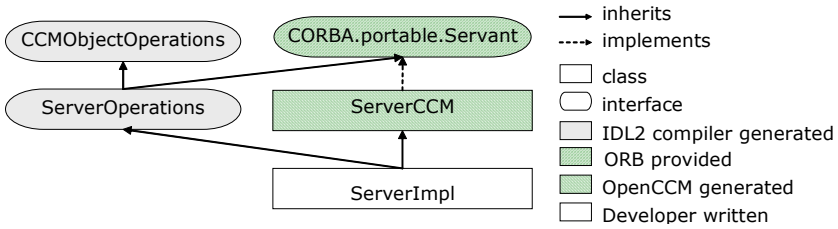

Figure 3 *Component object augmented structure for a component type called Server*

This component object structure is obtained as the result of a complex compilation chain involving an IDL3 (extended IDL) to IDL2 mapping phase and an augmented inheritance structure generation phase. The IDL2 interface generated during the mapping phase allows the use of a standard IDL compiler and thus ensures CORBA2 compatibility. It contains the component's business interface description (a fusion of all the IDL3 component interfaces as CORBA2 objects cannot implement multiple interfaces) as well as additional methods for component management.

In the case of the facet/receptacle IDL3 description[1] of our agenda application, the resulting IDL2 mapping specification is the following.

```
interface ManageReservations {.. Figure 1 business code};                    // Does not change
interface Client : CCMObject {  …Figure 1 business code                      // Component Client
        void connect_to_server(in ManageReservations connection) raises(...);  // Additional methods
        ManageReservations disconnect_to_server() raises(...);
        ManageReservations get_connection_to_server();
};
interface Server : CCMObject { ..Figure 1 business code                      // Component Server
         ManageReservations provide_for_clients();                           // Additional methods
};…
```

Figure 4 *IDL2 facet/receptacle description of the agenda*

Components are represented by interfaces extending the OpenCCM predefined `CCMObject` interface, responsible for providing all generic introspection operations. Ports are represented by *specific port management methods* which are part of the additional methods mentioned before. The `get_connection_to_server` method is a `Client's` introspection operation and returns the reference to the `ManageReservations` receptacle. This is the reference manipulated by a client when invoking the `Server` component. The rest of the additional operations are used for connection management in the deployment model and are discussed in the next section.

3.2.3 Deployment model implementation in OpenCCM

[1] In the following discussions we focus on synchronous communications. In fact, in OpenCCM, asynchronous interfaces (*push* event model) are represented by synchronous ones [15].

In OpenCCM, application deployment is done by a deployment program which includes statements for component archives installation, for component instance creation, for component configuration and for port (component) interconnection.

Port interconnection is done using the automatically generated OpenCCM port management interface added to the CORBA component object. In the case of our agenda application, the port management interface (Figure 4) for the `Server` component contains the `provide_for_clients` method : an introspection operation returning the reference to the `ManageReservations` facet. The `Client's` port management interface contains (in addition to the `get_connection_to_server` introspection method) the `connect_to_server` and `disconnect_to_server` operations through which the client is respectively connected and disconnected to the server.

OpenCCM defines several classes providing a basic deployment environment manipulated through a simple API. A schematic Java deployment program for our agenda application is given in Figure 5. The basic steps include the choice of the deployment hosts (step 1), the installation of components' implementations (step 2), the creation of component instances (step 3), their configuration (step 4), the component interconnection using the introspection facilities (step 5) and finally, the application launching (step 6).

```
// (1) Obtain the component deployment servers
// ns is referencing the CORBA's NamingService
ComponentServer cs1 =
ns.resolve("ComponentServer1");
// (2) Install component archives
// s1_inst is referencing the installation factory of cs1
s1_inst.install("agenda", "./archives/agenda.jar");
// (3) Create components
//sh/ch is the Server's/Client's instance manager
Server s = sh.create();
Client c1 = ch.create();
```

```
// (4) Configure components.
s.name("The Server");
// (5) Connect client and server.
// (5.1) Get the port reference using introspection
ManageReservations for_clients =
                    s.provide_for_clients();
// (5.2) Establish connection
c1.connect_to_server(for_clients);
// (6) Configuration completion, launching
s.configuration_complete();
```

Figure 5 *OpenCCM deployment of the agenda*

4 Replication management in OpenCCM

The design choices allowing to respond to the needs of an adaptable replication management infrastructure are discussed in the first part of this section. This presentation is followed by a description of our experience in which we apply the principles of the proposed infrastructure to two replication/consistency scenarios for the agenda application.

4.1 Principle

As we have seen in section 2, replication management includes copy creation and placement control, as well as copy consistency management.

Given that copy creation and placement modify the global architecture of an application and that the application architecture is defined in the deployment model, replication configuration can be naturally specified in the deployment model. Replication configuration in the deployment model is described in section 4.1.1.

In our approach, we suppose that consistency between copies can be managed using specialized treatments executed before and/or after normal (business) method invocations. Consistency management implementation relies therefore on invocation interception. This interception is closely related to the component interface definitions given in the abstract model

as it is done at the interface level, separately from the component implementation. Consistency management configuration is described in section 4.1.2.

4.1.1 Replication configuration

By defining *what* and *where* components are to be deployed as well as *how* these components are to be interconnected, the CCM deployment model describes the initial architecture of an application. Even if to date, CCM does not consider reconfiguration (the *when* aspect), the deployment model is the right place where reconfiguration[2] actions should be defined.

Since replication configuration consists in creating and interconnecting additional entities in the application's architecture, replication configuration can be naturally specified in the deployment model. In fact, a deployment model including replication will have to specify the set of replicable components (*what*), define the most appropriate moment for replication (*when*) and the best copy placement (*where*). The definition of a dynamic copy creation is to be part of the future reconfiguration specification features of the deployment model.

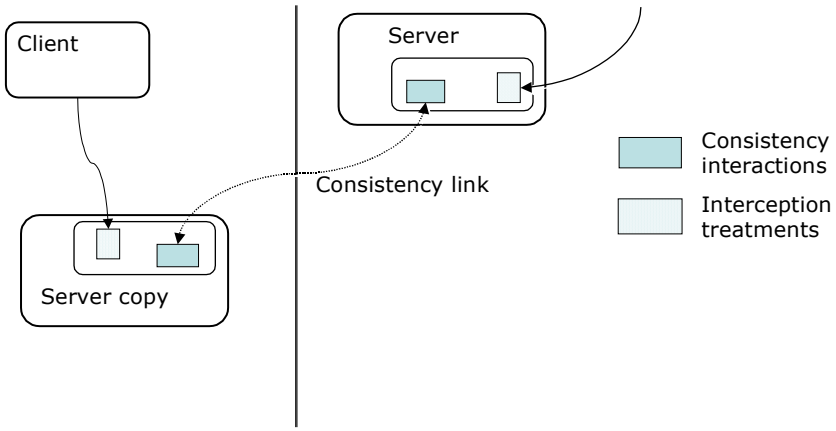

Figure 6 *A simple replication scheme*

Replication configuration in OpenCCM is added in the deployment programs. As these programs control component instance creation and interconnection, they are also given the responsibility of creating component replicas and connecting them to other components. In a cache management system for instance (Figure 6), the deployment program has to connect a client component to a local copy of a server component in replacement of the remote one. As the copies have to be kept consistent, there is a need for connections between the copies. In fact, following a specific consistency protocol, a deployment program has to establish consistency links between component replicas (examples of consistency links are described later).

4.1.2 Configuration of consistency management

Consistency management in a replicated *object* system generally relies on interception objects triggering consistency actions upon copy invocations. The use of interception mechanisms in a *component-based* system allows the integration of the consistency management aspect without modification of components' functional code. The consistency actions take the form of pre and

[2] We consider here architectural reconfigurations.

post treatments for the business method invocations 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 copies interconnections to propagate consistency actions. These connections are the consistency links mentioned in the previous section and settled during deployment. In the example of a cache management system, consistency actions may consist in the invalidation of a given copy and the acquisition of a fresh one which are typically implemented using consistency links.

Most consistency protocols require access to components' internal data. In the example of a cache management system, component's data need to be copied upon component's caching or update. The access to components' internal state may be based on global component capture/restoration or on application-specific selector/mutator functions. In our prototype, we preserve the component encapsulation principle by leaving the responsibility of implementing component state access primitives to the component developer. Consistency protocol implementations can thus rely on these primitives and ignore component implementation details.

The previously described interception objects and consistency links are the basis of consistency management. Their implementations depend of course on the specific consistency protocol chosen for a given application. The next section describes in more details the way interception objects and consistency links are generated and the roles they play 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
- Consistency links used to interconnect replicas
- Component accessor functions used to capture or modify internal data

Basically, an invocation in our replication-aware infrastructure (Figure 6) executes as follows. It is initially catched by an interception object and is afterwards propagated to a component copy. The interception object may trigger copy coordination by invoking consistency treatments possibly using component's data access functions.

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 for component replicas. The functional code of a component is managed in a separate object within the component which 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 interception object) define the nature of the interfaces between the component copies. The interfaces have to be described in IDL3 and are used to generate the final replicable component.

As mentioned in the previous section, the **component state capture/restoration treatments** are provided by the component developer who is aware of the component's specific semantics. In the case of Java components, we provide a default implementation based on Java Serialization [24].

The generation of replication management code is currently done by hand by extending the IDL3 component definitions, by providing an implementation of the consistency protocol and finally by adapting the application deployment program. Tools are under development, which will automate the generation of a replicable component associated with a particular consistency management protocol, and will help describing a replication architecture in the deployment model. In fact, the generation of a replicable component is pretty systematic as shown on Figure 7.

In Figure 7, both the component to be replicated and the consistency protocol to be applied are represented by their IDL3 definitions and their implementations. The consistency protocol's IDL3 definition declares the interfaces of the consistency links involved in the component copies coordination. The protocol implements these consistency interfaces as well as the component's interfaces in order to intercept the component's 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 steps:

- Component basic implementations: a developer provides the components' business code. If a component is to be replicated, he will need to provide primitives for state capture and restoration.
- Consistency protocol implementation: a replication expert implements a given replication/consistency scenario. The implementation is to be accompanied by corresponding IDL3 interfaces.
- Replicable component generation: the two IDL3 definitions (of the component and of the consistency protocol) are merged, and associated with both implementations to generate the component's final implementation.
- Replicated architecture deployment: after completion of the code generation phase, the application deployment is programmed. Components and component copies, as well as component interconnections including the needed consistency links are explicitly created in the deployment program.

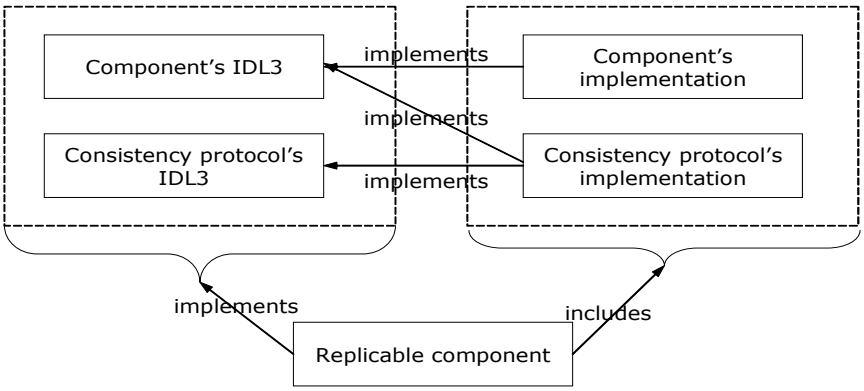

Figure 7 *Generation of a replicable component*

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 one implements a caching system.

4.3.1 Disconnection protocol

In the case of the disconnection protocol, we proceeded as follows. The agenda's components' implementations remain the same apart minor modifications in the `Server` in order to make it `Serializable` and to implement a default state management procedure. In the present description, we assume that the component provides two primitives granting access to the component's internal data i.e. `State captureState()` and `void restoreState(State state)` `State` being the type of the data). The IDL3 and generated IDL2 definitions for this component have already been given in Figure 1 and Figure 4.

The consistency implementation in this scenario distinguishes between master and slave server copies. 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 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 IDL3 definition for the consistency protocol is the following:

```
component Server… {
        attribute Role role;
        provides DisconnectionPtcl for_disc;
        uses DisconnectionPrtcl to_disc;
…}
interface DisconnectionPtcl {
        void makeCopy();
        void reconcile();
        State getState();
        void pushLog( in Log log);
};
```

Figure 8 *IDL3 definition for the consistency protocol*

Upon disconnection, the reconfiguration program creates a slave copy and invokes the *makeCopy()* method. This method calls the *getState()* method on the master and updates the slave's data before the disconnection gets effective.

Upon reconnection, the reconfiguration program invokes the *reconcile()* method which re-synchronizes the data. The log (*Log* interface) is propagated to the master using the *pushLog()* method.

Figure 9 shows a skeleton of the consistency protocol implementation in which the component object behaves as an interception objects. It holds a reference to the actual implementation of the component and forwards method invocations (e.g. *addReservation*). In this method, if the current component is a disconnected copy, the reservation has to be registered in the log for further reconciliation.

```
public ServerImpl() {
        if (role.isMaster()) realObj = new ServerActualImpl ();
        else log = new SimpleLog();
 }
public void addReservation(Reservation reservation) {
        realObj.addReservation(reservation);
        if (role.isSlave()) log.put(reservation);
}
public void makeCopy() {
        realObj = to_disc.getState();      //to_disc defines the port connected to the master, Figure 8
 }
public void reconcile() {
        to_disc.pushLog(log);
 }
public State getState() {
        return realObj.captureState();
}
public void pushLog(Log log) {             // rebuild state from the log …
        realObj.restoreState(state);
}
```

Figure 9 *Implementation of the consistency protocol*

Bellow (Figure 10) is the deployment program which configures the architecture of the application allowing disconnections.

In the first step, the server and client components are created. The client and the server are interconnected in step 2. Upon disconnection (step 3), a slave copy of the server is created. The connections are then updated (step 4) and the client is connected to the local copy of the server. The *makeCopy()* method (step 5) is invoked on the slave server in order to update its internal state. Upon reconnection (step 6), the *reconcile()* method is invoked in order to synchronize the two copies of the agenda, and the client is reconnected to the master server.

```
// (1) Create components
srv = SFactHost2.create();
srv.role(Master);
clnt =CFactHost1.create();
// (2) Connect client and server
ManageReservations for_clients =
srv.provide_for_clients();
clnt. connect_to_server(for_clients);
// (3) Disconnection
// Create a copy
copy = SFactHost1.create();
 copy. Role(Slave);
// (4) Update connections
clnt.disconnect_to_server();
for_clients = copy.provide_for_clients();
clnt. connect_to_server(for_clients);
DisconnectionPrtcl for_disc = srv.provide_for_disc();
copy.connect_to_disc(for_disc);
// (5) Update the copy before disconnection
copy.makeCopy();
// Effective disconnection
// (6) Reconnection
copy.reconcile();
clnt.disconnect_to_server();
ManageReservations for_clients = srv.provide_for_clients();
clnt. connect_to_server(for_clients);
```

Figure 10 *Deployment program for the disconnection scenario*

4.3.2 Caching protocol

We have also experimented with a caching system for the agenda application. This caching system allows to deploy copies of the server component on several client machines and to keep consistent portions of the agenda database. We have implemented a version of the *entry consistency protocol* [1], which follows a multiple-readers/single-writer protocol.

The implementation of the consistency protocol is very close to the one implemented in the Javanaise system [9]. Each method of the agenda server component is associated with a

component locking policy, depending on the method's nature (read/write) The interception treatment ensures that a consistent copy is cached before forwarding the invocation to it.

The deployment program specifies the sites where caching should be applied. 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 deployed architecture is shown on Figure 11.

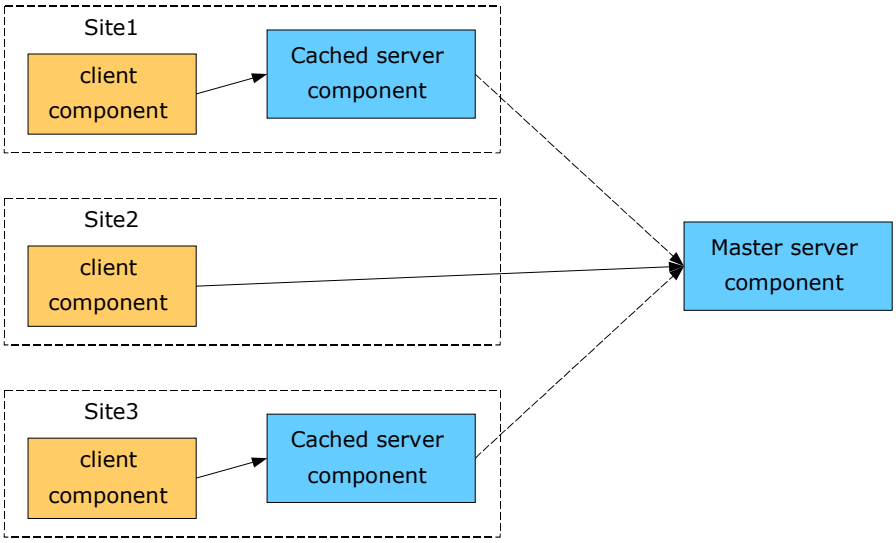

Figure 11 *Deployed architecture for the entry-consistency scenario*

The consistency links between replicas implement an interface (Figure 12) which defines operations for fetching (in read or write mode) an up-to-date component copy and for copy invalidation. There are actually two consistency links between a client and the server: one is used by the client in order to fetch copies (in read or write mode), the other is used by the server in order to reclame copies and locks held by the clients.

```
//consistency link from client to server
interface client2server  {
    State lock_read();
    State lock_write();
    void reduce_lock(State s);
    void augment_lock();
}
```

```
// consistency link from server to client
interface server2client {
    State reduce_lock();
    void  invalidate_reader();
    void  State invalidate_writer();
 }
```

Figure 12 *IDL3 definitions for a simple caching system*

In this prototype, the unit of consistency is the whole agenda, but it is be possible, with an adequate component capture/restoration interface, to manage finer grain consistency.

5 Lessons learned and related work

5.1 Lessons learned

This experiment allows us to draw several lessons learned.

We have shown that it is possible to manage replication as an adaptable non functional property in a component-based system. Our integration of replication management in OpenCCM is adaptable since it is possible to associate different replication and consistency policies with the same component-based application. It can be achieved in a non functional way since it does not

require any modification to the application business code. We have experienced with an exiting application (an agenda) for which we implemented two replication policies, one providing caching on client machines in order to reduce access latency, and one allowing disconnected clients to use a local copy and to reconcile copies upon reconnection.

We have identified two places where the non functional integration of replication management takes place. The first place is at the level of the interfaces of components. We use interception objects in order to capture invocation events and to trigger consistency actions. These interception objects could be replaced by adaptation mechanisms at the container level when they are available (many research groups are working on adaptable containers, both in the EJB and CCM environments). The second integration place is the deployment model. The deployment model is the place where the architecture of the application is described. Therefore, since replication impacts the architecture of the application, it logically has to be described in the deployment model.

OpenCCM provides port management functions. More precisely, these functions allow to get references to facets and to assign a reference to a receptacle. They are mainly used in deployments programs in order to build applications architectures. At the moment of writing, EJB does not provide an similar functionality. We believe that this is one of the most interesting functions of OpenCCM, especially for non functional replication management.

5.2 Related work

There are two major axes for comparison of our experiments to related research work: the first considers replication adaptation efforts done in component-oriented middleware and the second treats investigated adaptation mechanisms in the CORBA domain,.

Although research on components has been done for quite long now, component-based middleware has appeared only recently. Commercial products are either object-based or implement the EJB specification. Given the youth of the CCM model, prototype implementations are just starting to appear. Experiments with adaptation in component-based middleware are in consequence a very recent issue. In our knowledge, the presented work is the first experiment with replication adaptation in the CORBA component model. There are works treating replication on the EJB server level but these approaches apply the standard solution of fixed replication based on replication of the underlying relational databases. There are of course experiments with configurable replication. We can cite works on distributed shared memory [3] supporting several consistency protocols. Research on mobile disconnectable databases like Bayou [6] has introduced solutions of optimistic peer-to-peer consistency management with possibilities of application-specific reconciliation politics. Object-based research like Globe [21] or CORBA caching [4] have experimented even with non-functional implementations for replication configuration. Nevertheless, all these works provide domain-specific solutions and do not consider the component-based approach.

CORBA-centered works on replication are numerous. They divide in two major groups: works interested in fault tolerance and works on caching. In our knowledge there are no works trying to reconcile the two issues. Projects exclusively interested in fault tolerance basically apply the approach specified in the Fault Tolerant CORBA standard [20]. By modifying the ORB and the reference management, they introduce group references and invocation treatment based on multicast communication. Adaptation is more present in existing caching solutions. The CASCADE [4] project for example, uses the CORBA interceptors in order to insert different caching strategies. However, this solution may turn out to be non CORBA compliant as

interceptors which are central to this approach are to undergo major modifications in the CORBA standard. Flex [13] is a CORBA-centered project which is close to our work due to the use of object subclassing and of object-personalized state capture (stringification) in order to introduce different cache management policies. Yet it does not provide reference management allowing disconnection of the cache from the main server and remains an exclusively cache management project.

6 Conclusion and future work

We have investigated the integration of replication and consistency management in a component-based platform. We have proposed and implemented an infrastructure allowing to configure replication aspects in a non functional way. For our experimentation, we have used the first Java-based implementation of the CORBA component model: OpenCCM. The infrastructure we propose integrates well in the design of OpenCCM, as it does not modify the OpenCCM implementation and it remains CORBA2 compliant. Our design uses interception objects (which could be replaced by adaptable containers when OpenCCM provides it) and it exploits the CCM deployment model in order to describe an application replication policy.

We have defined the procedures for using this infrastructure and have shown its application in two scenario cases: the first implementing a caching system with entry consistency management and the second a simple disconnection management.

An immediate perspective of this work is to provide the tools which would allow automatic generation of the replicable component. Even if we have described the way replication can be added to an application in a non-functional way, most of the integration work is done manually and can be automated. Also, a tool that helps transforming the deployment program could be provided. It would allow specifying a replication policy in an application architecture and would generate the corresponding deployment program.

Another interesting perspective of this work is the investigation of the way this infrastructure can be applied to other component models. Candidates are notably COM and EJB but also more abstract models like ODP [12].

Finally, replication is only one system aspect and CCM one particular component model. The work presented in this paper is part of a broader project which aims at implementing a generic and reflexive component-based middleware. It should be generic in the sense it should allow to encapsulate different types of components (EJB, CCM ...). It should be reflexive in the sense it should allow adaptation for transparently integrating non-fonctional system aspects (persistence, security, replication ...).

References

[1] N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. *The **Midway** Distributed Shared Memory System*. In Proceedings of the 38th IEEE Computer Society International Conference, pages 528--537. IEEE, February 1993.

[2] A. Birrell, B. Nelson. *Implementing Remote Procedure Calls*. ACM Trans. on Computer Systems, 4(1):39-59, February 1984.

[3] J.Carter. *Design of the Munin Distributed Shared Memory System*. Journal of Parallel and Distributed Computing, 29(2):219-27, 1995

[4] G. Chockler, D. Dolev, R. Friedman, and R. Vitenberg. *Implementing a Caching Service for Distributed CORBA Objects*. In Proceedings of Middleware '00,1-23, April 2000

[5] 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, volume 1175 of Lecture Notes in Computer Science, 1996

[6] A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, et B. Welch. *The Bayou Architecture: Support for Data Sharing Among Mobile Users*. Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications, 2-7, December 1994.

[7] G.Eddon, H.Eddon. *Inside Distributed COM*. Microsoft Press, 1998

[8] Pascal Felber, Rachid Guerraoui, and Andre Schiper. *Replication of CORBA Objects*. In S. Krakowiak and S.K. Shrivastava, editors, Recent Advances in Distributed Systems, volume 1752 of LNCS. Springer Verlag, 2000.

[9] D. Hagimont, F. Boyer. "A Configurable RMI Mechanism for Sharing Distributed Java Objects". *IEEE Internet Computing*, Vol. 5, 1, pp. 36-44, Jan.-Feb. 2001

[10] R. Hayton, A. Herbert, et D. Donaldson. *Flexinet: a flexible, component oriented middleware System*. SIGOPS'98, Sintra, Portugal, September 1998

[11] Inprise. *VisiBroker 4.5 for Java. User Guide*, 2001. http://www.inrise.com

[12] ITU-T Recommendation X.901 | ISO/IEC 10746. *Reference Model - Open Distributed Processing*.1995. http://www.dstc.edu.au/Research/Projects/ODP/standards.html.

[13] R. Kordale and M. Ahamad. *Object caching in a CORBA compliant system*. USENIX Computing Systems, 9(4):377404, Fall 1996

[14] C. Marchetti, M. Mecella, A. Virgillito, R. Baldoni, *An Interoperable Replication Logic for CORBA Systems*, in Proceedings of the 2nd International Symposium on Distributed Objects and Applications (DOA 2000), pp. 7-16, Antwerp, Belgium, 21-23 September 2000.

[15] Raphaël Marvie, Philippe Merle, Jean-Marc Geib, Mathieu Vadet, *OpenCCM : une plate-forme ouverte pour composants CORBA*, 2ème Conférence Française sur les Systèmes d'Exploitation, Paris, France, April 2001

[16] Object Management Group. *CORBA Components: Joint Revised Submission*. Août 1999. OMG TC Document orbos/99-07-{01..03,05} orbos/99-08-{05..07,12,13}

[17] Object Management Group. *The Common Object Request Broker: Architecture and Specification, Revision* 2.4. OMG October 2000. ftp://ftp.omg.org/pub/docs/formal/00_10_02.pdf

[18] OOC. Orbacus 4.0.5 for C++ and Java User Guide, 2001. http://www.ooc.com

[19] OpenORB. *OpenORB 1.0.1 User Guide*, 2001. http://www.openorb.org

[20] Chris Smith, *Fault Tolerant CORBA* (Fault tolerance joint initial submission by Ericsson, IONA, and Nortel supported by Alcatel), ftp://ftp.omg.org/pub/docs/orbos/98-10-10, October 20, 1998.

[21] M. van Steen, P. Homburg, and A.S. Tanenbaum. *Globe: A Wide-Area Distributed System*. IEEE Concurrency, January-March, 1999, pp. 70-78

[22] Sun Microsystems. *Enterprise Java Beans Specification Version: 2.0*. Sun Microsystems. 2000. http://java.sun.com/products/ejb/.

[23] Sun Microsystems. Java™ Remote Method Invocation Specification. Sun Mycrosystems, 1998

[24] Sun Microsystems, Inc. *Java ™ Object Serialization Specification*, Sun Mycrosystems , 1996

[25] A.Tanenbaum. *Distributed operating systems*. Prentice_Hall, 1995

[26] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. *Understanding replication in databases and distributed systems*. In Proceedings of 20th International Conference on Distributed Computing Systems (ICDCS'2000), pages 264-274, Taipei, Taiwan, R.O.C., April 2000.