An Architectural Approach to Replication Configuration

Vania Marangozova, Daniel Hagimont

INRIA Rhône-Alpes, 655 avenue de l'Europe, Montbonnot 38334 St Ismier cedex, France
{Vania.Marangozova, Daniel.Hagimont}@inrialpes.fr

Abstract. Replication is a traditional means for guaranteeing service availability in the distributed computing domain. However, existing solutions manage the inherent replication and consistency complexity by limiting the problem to the specific needs of a given application domain and of a given execution environment. The approach has two major drawbacks : (i) replication solutions are most challenging to reuse and (ii) applications cannot be deployed in different execution environments (with different replication and consistency requirements) without major changes to their structure and code.

In this paper we propose an architectural approach to replication. Replication and consistency protocols are conceived as component-based architectures capturing the protocol logic and providing the corresponding implementation code. Defined in an application-independent manner, these architectures can be applied to different application scenarios. Moreover, their integration in component-based applications is done in a non-intrusive way i.e. without component business code modifications. Using a cache management protocol and a calendar application as examples, the paper describes the solution principles, presents our prototype implementation and discusses performance issues.

Keywords: replication, non-intrusive configuration, component architectures

1 Introduction

Widely used in the areas of fault tolerance [20], caching [4] and disconnection management [5], replication is a traditional means for enhancing service availability and performance. However, it is faced today with an important multiplication and diversification of the execution environments and platforms. To respond to the availability requirements of applications deployed in such different contexts, replication needs to be managed in a configurable way.

Existing consistency and replication solutions fail to provide appropriate configuration. In fact, their specificity, as well as close integration in the application architecture, is a major difficulty to replication solutions' conception, maintenance and reuse.

The past few years have seen the emergence of component-based programming which promotes the separation of services' business and administration aspects [15]. As replication can be seen as a particular administration aspect, the component approach does provide a generic framework for replication integration and configuration. However, it does not really answer the questions of replication management organization i.e. of application-independent replication solution conception, of non-functional application-configuration (replication integration without business code modifications) and of reuse.

In this paper we propose an architectural approach to replication. The idea consists in applying the concepts of components and component architectures [19], usually used for business application construction, to the replication and consistency aspects. Viewed as a specific component-based application, replication management is conceived and programmed in an application-independent manner thus allowing for integration and reuse in different applications. We describe our prototype implementation and discuss application and protocol architectures, programming and interactions.

The organization of this paper is as follows. Section 2 details the nature of replication and consistency treatments and their relation to component architectures. Section 3 presents the chosen component model and gives some implementation details. Section 4 is dedicated to the principles and the implementation of replication and consistency architectures. Section 5 discusses the interaction model between application and replication architecture. Section 6 discusses related work and Section 7 concludes the paper.

2 Non-intrusive Replication through Aspect Architecturing

Our work has two main objectives both of which are related to reuse and facility of conception and configuration. The first objective concerns applications and focuses on business component reuse (without code modification) in different deployment environments. It considers the possibility to attach different replication and consistency protocols to the same application. The second objective is symmetric: it aims at reuse of replication solutions (without code modifications) in different application contexts. It considers the possibility to use the same protocol with different applications.

In order to easily configure both application and replication while maintaining an operational system, we separate the implementations of the two aspects and define an interaction model.

Separation between the replication and application aspects is component-based. The component paradigm pays a particular attention to applications' administration and promotes the separation between the components' business logic implementations and their administration code. Being a particular administration aspect, replication can take advantage of the paradigm's characteristics.

The application-replication interaction model is based on architecture models. Replication and consistency are closely related to architecture issues: replication creates and places additional components (copies) while consistency establishes additional component interactions (consistency maintenance). Controlling and configuring replication and consistency is in consequence equivalent to manipulating and configuring application architectures.

Application architectures [19], initially introduced in the area of software engineering and increasingly used in all computing domains, have one main advantage which is *control*. In fact, architectures open up the previously hidden application structures and facilitate design complexity management, component reuse and reconfiguration [6].

While architecture seems to be accepted as a natural application characteristic, administration aspects (e.g. replication and consistency) are in general complex low-level mechanisms and have major difficulties to be considered from a modeling (architectural) point of view. However, the benefits of administration architectures are numerous and analogous to the ones in the application context. In the case of replication and consistency, architectures provide facilities for protocol conception complexity management (currently prohibitive), for protocol entities reuse (currently quasi-impossible) and for protocol reconfiguration. Moreover, as replication and consistency are already articulated in terms of application architectural items (application components and interconnections), architecturing the aspects will render even more explicit the control points needed for the interaction model between a protocol and an application.

The following sections describe in detail the application and replications architectures as well as their interaction model.

3 Architecturing the Application Aspect

We have implemented a minimal component model concentrating on the necessary features for architecture description and manipulation of replication and consistency management. This section describes the model's principles and implementation.

3.1 The Component Model

Components in our model are entities characterized by the set of their *provided* and *required* interfaces. These interfaces specify in an explicit way the entry and exit communication points of a component. The communication points are separate entities intercepting all component interactions.

Components have global unique references. These references give access to an introspection interface providing information on components' type and execution state.

Entry and exit communication have references typed with the corresponding required or provided interfaces. These references are obtained through introspection and are used for component interconnection. Connections are established between an exit point and an entry point typed with the same interface (required in one case, provided in the other).

Applications in our model are sets of interconnected components. Their architecture is defined during an explicit deployment phase responsible of creating, placing and connecting components. Connections are set and may be reconfigured during execution using a specified connection interface.

Let us consider an agenda management example application in this model (Fig. 1).

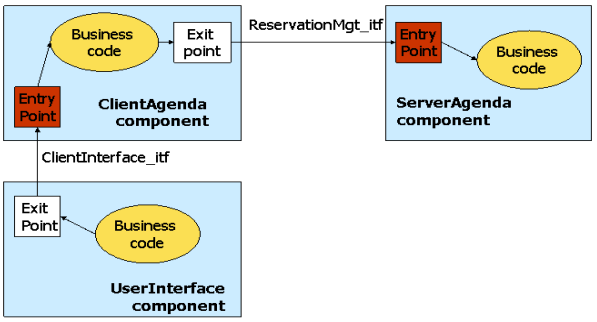

Fig. 1. A simple agenda application

In this application, users can connect through a graphical interface to an agenda server and register, edit or remove rendezvous from their planning's. There are three components: `UserInterface` which represents the graphical interface, `ClientAgenda` which accepts and transforms requests from the interface before passing them to the server and `ServerAgenda` which is responsible of all reservation management. The required and provided interfaces are the following. The `ReservationMgt_itf` interface is provided for the server and required for the client. The `ClientInterface_itf` is required for the graphical interface and provided by the client.

If we consider more in detail the `ClientAgenda` structure (Fig. 2), we have the global component reference which can introspect the structure and report the existence and the connection state of the two communication points corresponding to the provided `ClientInterface_itf` and required `ReservationMgt_itf` interfaces. The business code entity encapsulates the client implementation.

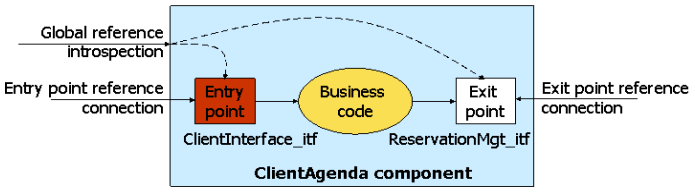

Fig. 2. A component structure

3.2 Implementation

Our Java-based prototype provides four basic entities implementing the component model: `Container`, `Component`, `Stub` and `Skeleton`. The `Container` class represents the global component structure and contains introspection information as well as references to all communication points and to the business implementation object. Exit and entry communication points are represented by `Stub` and `Skeleton` entities which implement the corresponding required or provided interfaces. `Component` is the super class for all business component implementations. Instantiating a `Component` triggers the creation of a corresponding container with all the needed stubs and skeletons.

As it has already been mentioned, communication points play the role of interceptors (Fig. 1). The outgoing business client invocations, for example, are intercepted by the client stub, sent to the server (possibly through the network), intercepted by the server skeleton and finally passed to the servicing code. The performance of such an invocation scheme are discussed later in this paper.

To implement a component in our prototype one needs to provide a XML descriptor of the component's communication points and to create a class providing the business method implementations which extends the predefined `Component`. All stub and skeleton structures are generated and automatically integrated in the container.

The schematic implementation of the `ClientAgenda` component is given in Fig. 3.

```
public class ClientAgenda extends FAR.Component implements ClientInterface_itf {
   //reference to a ServerAgenda, an exit point
   public ReservationMgt_itf my_server;

   //component code, reference used as normal Java reference
   public okButton() {…
      mon_server.addReservation(…);
   }}
```

Fig. 3. A schematic component implementation

The descriptor for this component is given in Fig. 4. It gives the information about the component communication points, their interfaces (`ResrvationMgt_itf and ClientInterface_itf`) and the name under which they are known in the component's code (`my_server` and `buttons`).

```
<component ClientAgenda><points>
   <out>
     <interface>ReservationMgt_itf</interface><name>my_server</name>
   </out>
   <in>
     <interface>ClientInterface_itf</interface><name>buttons</name>
   </in>
</points></component>
```

Fig. 4. A component XML descriptor

Application deployment is done using a basic API which allows to connect at a distance to a component execution environment and to create and to interconnect component instances. A simplified deployment program for the agenda application is given in Fig. 5.

```
//get the deployment control points of two distinct execution environments
ctxt1 = <naming_service>.get("DeploymentServer1");
ctxt2 = <naming_service>.get("DeploymentServer2");

//instance creation: client and server execute in contexts ctxt1 and ctxt2
client = ctxt1.create(ClientAgenda,…);
server = ctxt2.create(ServerAgenda,…)

//connection of the client's communication point (my_server)
//to the server's corresponding entry point (real_server)
client.connect(my_server, server, real_server)
```

Fig. 5. A deployment example

Although simple, the chosen component model does provide the characteristics needed for replication and consistency management. The model provides facilities for defining application architectures (using the deployment phase), for component introspection, for connection control and reconfiguration.

4 Architecturing the Replication Aspect

Before introducing replication and consistency architectures, we discuss non-intrusive integration of replication and consistency in the presented application model and explain why a monolithic management approach is not sufficient.

4.1 From monolithic to architectured management

Our objective of non-intrusive replication and consistency integration in component-based applications (integration without modification of their business code) implies that neither the components' business implementations (Fig. 2), nor their business interfaces can be changed. The integration has to be in consequence done in the structure encapsulating the business implementation i.e. the container. This approach is moreover the one promoted by the component paradigm.

Adapting the container structure in order to manage replication and consistency may be done in two ways: by adding new replication and consistency dedicated entities in the container or by adapting the

already existing ones i.e. the communication points. The latter possibility is based on the hypothesis that replication and consistency can be managed using treatments triggered before and after business invocations. It can be considered as a reflection-based approach [16] where business method invocations are reified i.e. their treatment is delegated to a specific control level (Fig. 6).

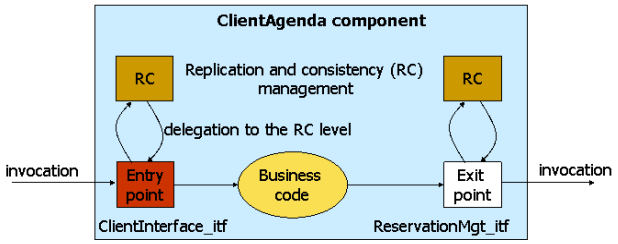

Fig. 6. Monolithic integration of replication and consistency

This has been our first approach to managing replication and consistency in a non-functional way. The most simple way to implement it [17], has been to augment the existing stub and skeleton structures in order to integrate the corresponding replication and consistency treatments. In fact, the two levels shown in Fig. 6 have been implemented as a single one. Such a solution does prevent additional invocations and assures a better performance than a two level approach. However, its shortcomings are numerous. First of all, the assimilation of the replication and consistency management entities within the application stubs and skeletons, means that there is no possible protocol modification without recompilation of these structures. This is contrary to our aspiration to configure already existing applications. On the other hand, protocol management involves replication and consistency dedicated interactions. With a single management level, these interactions are to be intercepted and treated by the same entity which is responsible of intercepting and dispatching business invocations. There is a problem of concurrency and of mutual disturbance.

Deciding to separate the two types of treatments does not resolve all problems. Replication and consistency continue to be implemented following a monolithic approach: the facets of copy creation and placement, of state manipulation and of consistency maintenance are all mixed up. The approach allows the implementation of simple protocols but it falls short when it comes to defining a framework for protocol implementation, reuse and reconfiguration. This is the major argument in favor of replication and consistency architectures.

4.2 Architecture Model for Replication and Consistency

The principal idea in defining replication and consistency architectures is to consider that replication and consistency management is *itself* an application. This issue is discussed in Section 4.2.1 and applied to an example in Section 4.2.2.

4.2.1 Principle

In a component-oriented approach, a protocol can be viewed as a component-based application where components and their interconnections are specific to the given replication and consistency domain. Protocol interactions can be organized in three groups: replication-dedicated, state-management oriented and consistency-dedicated. *Replication-dedicated interactions* are responsible of carrying out all actions related to copy creation, placement and destruction. Such interactions will typically be in charge of the notification of a group of copies upon a copy creation. *Consistency-dedicated interactions* are in charge of the synchronization of copies meant to provide similar functionalities according to some correctness criteria. Synchronization is based on exchange of state or operation log information. *State-management interactions* handle state manipulation involved during copy creation or synchronization.

Modeling a protocol as a component-based application requires the existence of an appropriate component model which responds to the following requirements. It has to allow easy protocol entity creation, configurable protocol construction (possibility to decide of the number, type, placement and interconnection of protocol entities) and protocol reconfiguration (interaction modification, component replacement, etc. in existing protocol architectures).

Our component model (Section 3.1) actually satisfies all of the above requirements. It facilitates protocol entity conception as it concentrates on the implementation of the functional logic of components. It allows configurable protocol construction through its deployment facilities. During deployment, it is possible to instantiate different protocol architectures with different sets of components and interconnections. Finally, the explicit manipulation of the architecture allows for architecture reconfiguration.

4.2.2 The Entry Consistency Protocol Example

Let as consider an example of a replication and consistency protocol and see how this protocol can be architectured using our component model i.e. using components with explicit communication points and interconnections. The chosen protocol, similar to the one in [8], implements a master-slave replication scheme and an entry consistency management. In it, client entities issue read or write requests on objects which are managed in server entities. When a client needs to make a call on an object, it first verifies whether the object is in its cache. If this is not the case, it triggers the creation of a local copy of the object (implying the copying of the object's state). If there is a local copy, its consistency state is verified and if the local version is not the most recent one, it is obtained from the server. In order to operate on a local object, a client needs an authorization. Authorizations are given by servers and are represented as read and write locks. Each operation is thus preceded by a corresponding lock acquisition and followed by a lock liberation. Lock liberation is not propagated to the server but is cached locally in order to optimize possible subsequent accesses. A local copy is invalidated by a server when another object demands a conflicting lock on the same object. Upon invalidation, servers update the object's state.

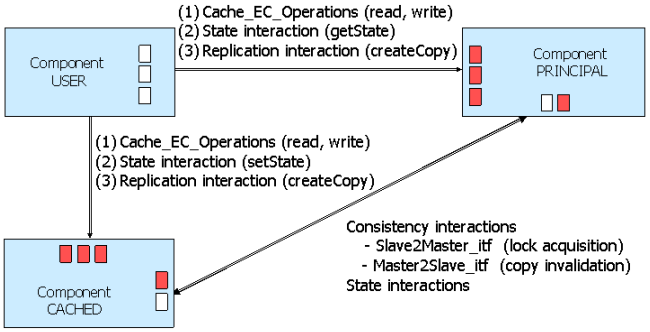

Fig. 7. Architecturing an entry consistency protocol

Architecturing the entry consistency protocol is done as follows (Fig. 7). There are three types of components: `USER`, `PRINCIPAL` and `CACHED`. The `USER` component corresponds to clients. As they issue read and write requests, this component has a read/write (`Cache_EC_Operations`) required interface. The `PRINCIPAL` component represents objects managed by server entities and therefore provides the `Cache_EC_Operations` interface as well as the `Slave2Master_itf` interface for lock management. It requires the `Master2Slave_itf` interface for copy invalidation.

In the protocol architecture, `USER` and `PRINCIPAL` are connected via their `Cache_EC_Operations` communication points. As the first `USER` request triggers the creation of a local copy (represented by a `CACHED` component), `USER` and `PRINCIPAL` are also connected through replication and state-management communication points. In fact, after the creation of a local copy instance, the `USER` is charged of getting the server's state (`getState` through the state link), initializing the copy (`setState` through the state link to `CACHED`) and notifying both of their mutual existence (replication links).

The `CACHED` component provides the `Cache_EC_Operations` interface to which the `USER` is connected after the copy creation (connection reconfiguration). It also provides the `Master2Slave_itf` interface which is the invalidation entry point for `PRINCIPALs`. It requires the `Slave2Master_itf` interface. Given that these consistency interactions involve updates and invalidations of copies, `CACHED` and `PRINCIPAL` are also connected through state-management links.

The described architecture is a generic one i.e. it concentrates exclusively on replication and consistency management and makes abstraction of all application-specific details: there is no description of the specific nature of a `PRINCIPAL's` object, there is no specification of the number of

PRINCIPAL instances and there is no limitation on the possible USER to PRINCIPAL connections. What this architecture defines is that a connection from a USER to a PRINCIPAL implies the USER's manipulation of the PRINCIPAL's object according to the entry consistency protocol. The application-independent nature of the architecture allows its reuse in different application contexts.

4.3 Implementation

As there is a common component model between applications and protocols, the implementation of protocol entities is quasi-identical to the one in the business application case. The differences are due to the specificity of the protocol interactions which introduce the definition of specific replication and state management interfaces and to the fact that protocol components do not have independent existence during execution but their treatments are integrated in the underlying business application structure. This is shown with the example of the CACHED component's implementation (Fig. 8). Its implementation class extends ProtocolComponent instead of Component. This class is not instantiated but is used to generate protocol-specific control entities. It implements the four provided interfaces (Master2Slave_itf, Cache_EC_Operations, ReplicationMgt_itf, and StateMgt_itf) corresponding to its entry points. Its exit point is represented as a class attribute (principal).

```
public class Cache_EC_CACHED extend FAR.RC.ProtocolComponent implements
          Master2Slave_itf, Cache_EC_Operations,
          ReplicationMgt_itf, StateMgt_itf,
          DeploymentConfiguration_itf
{
   //consistency exit point, reference to a principal
   public Slave2Master_itf principal;

   //implementing the protocol's Cache_EC_Operations
   public void read() {
      principal.lock_read(…); //obtain the lock
      …//actual read on the object
      …//unlock, cache the lock
   }
   public void write() { //similar implementation }

   //implementing the Master2Slave_itf consistency interface
   public void invalidate_reader() { … }
   public Object invalidate_writer() { … }
   …
   //implementing the ReplicationMgt_itf interface
   public Object createCopy(…) { … }

   //implementing the DeploymentConfiguration_itf
   public <comm_point>[] getDConnections(<component_type>)  { … }
}
```

Fig. 8. A schematic protocol component implementation

The last interface, DeploymentConfiguration_itf, actually gives information about the wanted deployment protocol architecture. The getDConnections method returns the set of communication points supposed to be connected to a component of type component_type. This is the place where a CACHED describes what communication points are supposed to be connected to corresponding points of a PRINCIPAL component. Together with the PRINCIPAL's getDConnections method, the default deployment interconnections between the two are described. This default architecture can be modified by an additional deployment descriptor.

5 Putting the Aspects Together

We have described in Section 3 the nature of our application architectures. We have applied their component-based approach and defined replication and consistency architectures. Both types of architectures are specified in an independent manner. To make them work together i.e. to be able to

configure an application with a given replication and consistency protocol, there is a need of a specification of an interaction model. The interaction model is discussed in Section 5.1. Its instantiation for the agenda application and the entry consistency protocol is described in Section 5.2. Implementation and performance details are given in Sections 5.3 and 5.4.

5.1 Principle

We have chosen to not instantiate directly the protocol architecture. This would have introduced the creation of an explicit meta-level to which all replication and consistency treatments would have been delegated. The resulting global architecture would have been a very heavy one.

We have preferred to define a correspondence between the application and the protocol entities allowing the generation of entities integrating the protocol-specific treatments directly in the application architecture. We have seen in Section 4.1 that non-intrusive replication integration is to be done at the container level using interceptors. The chosen technique pushes the approach a little bit further and integrates the protocol interactions into the application. The container-integrated entities correspond in fact to the communication points of the components defined in the protocol architectures. Some of them do act as interceptors for the business invocations while others do not.

Intuitively, the correspondence between the application and the protocol may be done at the component level i.e. one application component is related to one protocol component. However, such an approach will imply that all component's interactions are managed following the rules defined in the corresponding protocol component. For example, all outgoing invocations of a client component will be obliged to use cache management. This in turn will mean that the components whose entry points are connected to the client's exit points are also managed by the same protocol. In summary, a component-level correspondence results in a rather simplistic and constraining situation.

The correspondence is necessarily done between explicit architectural elements. If components are not appropriate, interfaces (entry and exit communication points) seem to be a good starting point. Attaching a protocol interface to a business one will imply the modification of the corresponding business interaction according to the protocol rules. Moreover, the thus established relation between the application and protocol communication points will identifies the replication and consistency interceptors for business invocations.

5.2 Example

If we consider our agenda example (Fig. 1) and the presented entry consistency protocol (Fig. 7), the association is rather simple. The objective of the protocol integration is to modify the `ClientAgenda` to `ServerAgenda` interaction in order to introduce cache management.

The entity to be cached is the server. Every operation from the `ReservationMgt_itf` interface is to be interpreted as a read or write operation and to be treated accordingly. The correspondence is in consequence between the `ReservationMgt_itf` and the `Cache_EC_Operations` (read/write) interfaces. A `ClientAgenda` will be assimilated to a `USER` protocol component while the `ServerAgenda` is a `PRINCIPAL` component (Fig. 9).

After the protocol integration, the `ClientAgenda` component's container is added the communication points defined in the `USER` protocol component. In the same way, the `ServerAgenda` finds itself augmented with the communication points defined in the `PRINCIPAL` component. As Cache_EC_Operations is mapped to ReservationMgt_itf and there is an established ReservationMgt_itf connection, the preprogrammed protocol deployment establishes the three corresponding connections (`Cache_EC_Operations`, `StateMgt_itf` and `ReplicationMgt_itf` interactions). There is also a link between the `ReservationMgt_itf` communication points and the `Cache_EC_Operations` ones which is set up for business invocation delegation.

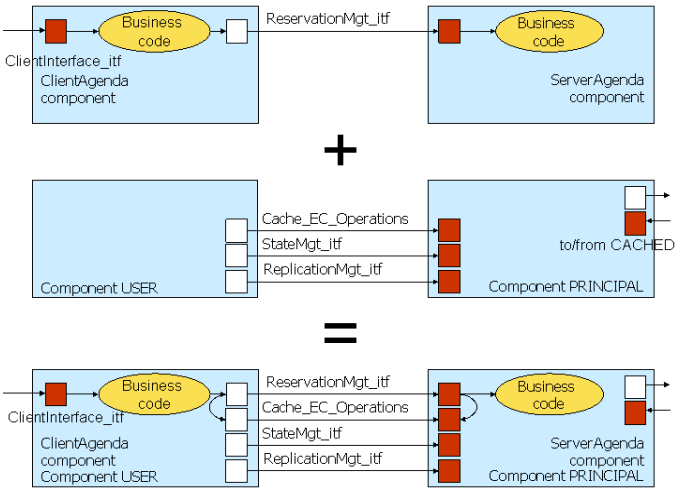

Fig. 9. Application replication/consistency configuration using a protocol architecture

5.3 Implementation

Configuring a component-based application with a given protocol requires the following. First, a correspondence like the one described in the previous section must be provided. The correspondence is currently given by a XML descriptor. In the agenda-entry consistency example, the descriptor is the following (Fig. 10).

```
<mapping>
   <application_itf>ReservationMgt_itf</application_itf>
   <protocol_itf>Cache_EC_Operations</protocol_itf>
      <operation>addReservation;write</operation>
      <operation>listReservations;read</operation>  …
</mapping>
```

Fig. 10. Descriptor of the correspondence between application and protocol interfaces

Using this correspondence, appropriate stub and skeleton entities are to be generated. The generation process uses the provided protocol generic implementation and specializes them in order to implement the corresponding (in terms of the established correspondence) application interfaces. This specialization prevents the use of costly generic invocations such as the Java's `invoke`. The generation may be anticipated or be done on the fly during deployment.

Fig. 11 shows the generated code for a skeleton attached to a `ServerAgenda` component.

```
class Cache_EC_CACHED_Mapping_Skel extends _RC_Skel_Impl
                                   implements ReservationMgt_itf {
 public boolean addReservation(Reservation p) {
   principal.lock_read(…); //code from the generic implementation, obtain the lock
   //actual read on the object, replaced with the application specific operation
   val=((ReservationMgt_itf)component).addReservation(p);
   //unlock, cache the lock
   return val;
 }…
}
```

Fig. 11. Generated protocol component skeleton for a specific application

Business method stubs and skeletons reference the attached protocol ones and vice versa. These references are initialized upon the configuration of a component with a protocol and modulate component interactions. Upon an ingoing business method invocation (Fig. 12), the corresponding skeleton verifies whether it has been attached a protocol (reference state). If this is not the case, the invocation follows its standard way. If there is an associated protocol, the invocation is delegated to the protocol control skeleton. This skeleton executes necessary replication or consistency actions,

possibly calls the business implementation, executes other protocol operations when this invocation terminates and returns the control to the application skeleton.

Upon an outgoing business invocation, there also is a protocol existence test. If a protocol is attached, the invocation passes through the protocol stub. The stub takes in charge the invocation if there is a need to pass additional replication/consistency information with the business invocation.

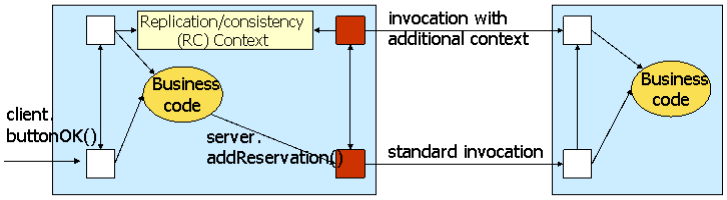

Fig. 12. Invocation management in the presence of a protocol

Protocol specific information may be updated and stored in the container by the protocol-dedicated stubs and skeletons (`Replication/Consistency Context`). In the entry consistency protocol this will typically be information about lock ownership.

The possibility to pass business invocation control over to the protocol stubs and skeletons implies that the application structures include an appropriate "hook". In fact, in our current implementation, this "hook" takes part of these generated structures even if the application is deployed without replication and consistency configuration.

The command to attach a protocol to an application i.e. to execute all of the above generation and integration operations is given during deployment. The deployment program for the agenda application given in Fig. 5 continues with the following instructions for protocol configuration (Fig. 13).

```
//attaching the protocol components to the application communication points
client.attachProtocol(Cache_EC_USER, my_server);
server.attachProtocol(Cache_EC_PRINCIPAL, real_server);
//triggering the pre-programmed protocol deployment
protocol_complete();
```

Fig. 13. Protocol configuration commands during deployment

After the completion of this protocol deployment, the application may execute following an architecture augmented with replication and consistency management.

5.4 Performance and Optimization Issues

The introduction of an additional interception level (the replication and consistency control stubs and skeletons) leads undoubtedly to poor performance. If a normal local Java call costs about 7ns, a local invocation in our system costs about 24ns i.e. is about four times longer. Distant calls also keep about the same ratio as compared to distant Java (RMI) calls. The latter is however mostly due to the dynamic computation of parameter types during invocation marshalling and to the sharing of communication channels. These results are natural given the fact that this first prototype has exclusively concentrated on engineering issues and not on performance. However, there is a way for performance improvement. In fact, it will be possible to use adapted stub and skeleton generators. The idea is to keep the application and protocol architectures during the conception and the deployment phases but to use code optimization like byte-code injection techniques [9] in order to obtain a single-level (with no unnecessary interception and invocation) execution structure.

6 Related work

The issues of *adaptation and of non-functional configuration* are major objectives of numerous platforms and different mechanisms are proposed at different levels. Language platforms like the aspect-oriented AspectJ [14] project are interested in easy source code manipulation and modification. AspectJ provides advanced means for code "weaving" which is related to our integration and

generation phase (Section 5). However, the proposed techniques consider much more complicated situations than the simple interaction on the components' surface (containers and interfaces).

Extension and reconfiguration are also considered by various system-level projects [11] but are of real actuality in the middleware domain. Numerous projects like Flexinet [7] concentrate on communication stack adaptation. This approach does sometimes consider replication and consistency management but in this case the proposed solutions are predefined and application-semantics independent. In contrast, our work concentrates on the possibility to provide different replication and consistency protocols which can integrate application-dependent aspects for consistency-management optimization.

The proposed two level (application and replication) architecturing approach is similar to projects working on reflexive systems [1]. Reflection may indeed be used for establishing the relation between these two levels but is no more than a particular technique. Moreover, the resemblance with the reflection approach is a result of our ascending approach (start from the replication and consistency needs and see how to organize the aspects) while existing reflection-oriented research tend to have the opposite methodology: start from the concept, provide meta-level architectures and only then try to apply to different test cases.

Configuring replication and consistency in a non-functional way at the container level follows the trend of component-based middleware research in which adaptable containers and non-functional integration of system management properties are a major issue. However, projects working on adaptable containers do in general consider aspects like persistence, transactions and security but not replication. When replication is considered, for example in some commercial products managing load balancing and clusters of servers [2], the solutions are hidden and do not allow a modification or replacement.

Configuration and adaptation of r*eplication and consistency* in particular is considered by a number of projects . We can cite works on distributed shared memory providing multiple consistency protocols [3] or on mobile databases projects introducing optimistic consistency and application-specific reconciliation [5]. CORBA-centered research is also very present with works on flexible caching [4]. However, the solutions proposed by the cited projects remain domain-specific. Our work aims at overcoming this limitation and provides a means for architecturing and configuring of protocols coming from different domains.

Replication and consistency configuration may be considered as particular administration aspects aiming at creating optimal application execution configurations. This makes our work close to administration management research. Some recent projects [21] propose to consider administration as a complex distributed application and to treat it accordingly. Our work specializes the approach and applies it in the particular case of replication and consistency.

7 Conclusion and future work

We have investigated an architecturing approach to replication and consistency management. We have proposed and implemented a component-based infrastructure allowing the easy conception and non-intrusive integration of replication. We have successfully implemented the proposed principles in a Java-based prototype and experimented with different applications and protocols. Our design is based on the assumption that replication and consistency management is one particular application and uses the same architecture model in both replication and application cases. The integration uses a correspondence between the application and protocol interfaces and is done at the components' container level during deployment. Thus, the replication configuration of an application is exclusively done through architecture adaptation and without business code modification.

The presented work is almost entirely dedicated to software engineering issues: we describe a method for implementing and integrating replication and consistency solutions. Although the architectural approach to replication and consistency seems promising we need to further investigate means for efficient implementation. Also, we do not have tools facilitating the protocol implementation. Just as there are middleware platforms meant to hide distribution complexity and to help application conception and development, there should be specialized middleware facilitating the replication and consistency conception and integration. An important perspective of this work is

therefore the experimentation with a panoply of protocols e.g. Rover [13], AFS [12], etc. and the definition of a replication and consistency dedicated framework. Another important issue to consider is the type of applications to which this approach is really advantageous. We have experimented mostly with database-oriented applications having passive components. Applications with active components may need more complex mechanisms demanding some business code modifications.

A long term perspective will be the consideration of different component models like Microsoft's COM, EJB, CCM or ODP and the definition of usage patterns allowing for non-functional replication and consistency configuration. Such patterns will take into account not only the particularities of applications and their execution environments, but also the characteristics of the used component model. This issue takes part in a broader project aiming at the implementation of a generic component-based middleware allowing the encapsulation of different component types and the non-functional-configuration of different system aspects (persistency, security, replication, etc.)

References

[1] G. Blair, G. Coulson, P. Robin et M. Papathomas, An architecture for next generation middleware. Proc. of Middleware'98. Sept. 1998.

[2] BEA WebLogic Server, BEA, 2002. http://www.bea.com/index.shtml

[3] J.Carter. Design of the Munin Distributed Shared Memory System. Journal of Parallel and Distributed Computing, 1995.

[4] G. Chockler, D. Dolev, R. Friedman, and R. Vitenberg. Implementing a Caching Service for Distributed CORBA Objects. Proc. of Middleware'00, 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. Proceeding of the IEEE Workshop on Mobile Computing Systems and Applications, 2-7, December 1994.

[6] Noël De Palma, Philippe Laumay and Luc Bellissard. Ensuring Dynamic Reconfiguration Consistency. Sixth International Workshop on Component-Oriented Programming,ECOOP 2001.

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

[8] D. Hagimont, F. Boyer. A Configurable RMI Mechanism for Sharing Distributed Java Objects. IEEE Internet Computing, Jan.-Feb. 2001

[9] D. Hagimont, N. De Palma. Removing indirection objects for non-functional properties. Proc. of

[10] International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas (USA), June 2002.

[11] J. Helander and A. Forin, MMLite: A Highly Componentized System Architecture. Eight ACM SIGOPS European Workshop, Portugal, Sept. 1998.

[12] L.B. Huston, P. Honeyman. Disconnected Operation for AFS. Proc. of the USENIX Mobile and Location-Independent Computing Symposium, Cambridge, MA, 1993.

[13] A. Joseph, A. deLespinasse, J. Tauber, D. Giord and M.Kaashoek. Rover: A toolkit for mobile information access. Proc. of the 15th Symposium on Operating Systems Principles, Dec. 1995.

[14] 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.

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

[16] P.Maes. Concepts and Experiments in Computational Reflexion. Proc. of ACM OOPSLA, 1987

[17] V. Marangozova and D. Hagimont. An Infrastructure for CORBA Component Replication. Proc. of the First International IFIP/ACM Conference on Component Deployment, Berlin, 2002.

[18] D.Schmidt, M. Stal, H.Rohnert and, F.Buschmann. Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects. Wiley & Sons, 2000.

[19] M. Shaw and D.Garlan. Software Architecture, Prentice Hall, 1996.

[20] A.Tanenbaum. Distributed operating systems. Prentice Hall, 1995.

[21] M. Wood, K. Marzullo, Tools for Distributed Application Management, Proceedings of the Spring 1991 European Conference, Tromso, Norway, May 1991.