RS2.7: an Adaptable Replication Framework

Stéphane Drapeau*, ** — Claudia Lucia Roncancio** — Pascal Déchamboux*  

* France Télécom R&D  
28, chemin du Vieux Chêne  
F-38243 Meylan  
{Stephane.Drapeau, Pascal.Dechamboux}@rd.francetelecom.com  

** Laboratoire LSR IMAG  
681, rue de la Passerelle  
F-38402 Saint Martin d’Hères  
{Stephane.Drapeau, Claudia.Roncancio}@imag.fr

ABSTRACT. The RS2.7 Replication Framework revisits the replication function in order to provide a component-based middleware support that can adapt to several kind of environment. It clearly identifies what minimal functions are relevant to replication: binding replicas between themselves, and synchronizing them in order to support the right levels of coherency. This paper focuses on the coherency issue. We analyse how this feature can be decomposed with respect to two dimensions: functional and scheduling. Playing with these two dimensions allows to provide different replication solutions by merely assembling RS2.7 components. A prototype of RS2.7 is operational and has been applied to a platform for interactive networked applications.

RÉSUMÉ. RS2.7 est un Canevas de Duplication redéfinissant la fonctionnalité de duplication, afin de fournir un intergiciel à base de composants pouvant s’adapter à différents environnements. Il identifie les fonctions minimales propres à la duplication : la liaison entre les copies et leur synchronisation afin d’obtenir la cohérence locale souhaitée. Ce papier se focalise sur les problèmes de cohérence. Nous analysons comment cette caractéristique peut se décomposer selon deux dimensions : fonctionnelle et structurelle. En jouant avec ces deux dimensions, il est possible de fournir différentes solutions gérant la duplication en assemblant simplement des composants conformes à RS2.7. Un prototype de RS2.7 est opérationnel et a été utilisé dans une plate-forme pour applications interactives en réseau.

KEYWORDS: Replication, consistency, coherency, functional decomposition, adaptable framework, component.

MOTS-CLÉS: Duplication, cohérence globale, cohérence locale, décomposition fonctionnelle, canevas adaptable, composant.
1. Introduction

Distributed computing requires mechanisms to ensure system availability and reliability, with different purposes in mind such as fault-tolerance or performance scalability (e.g., parallel computing). Replication is at the heart of solutions to all the preceding issues and is usually declined with many different flavours. The RS2.7 Replication Framework revisits the replication function in order to provide a middleware support that can adapt to several kind of environment. It clearly identifies what minimal functions are relevant to replication: binding replicas between themselves, and synchronizing them in order to support the right levels of coherency. Moreover, this paper analyses how coherency can be decomposed with respect to two dimensions: functional and scheduling. Our work is done in the context of the NODS project\(^1\) (Network Open Database Services) that aims at defining an open, adaptable architecture that can be extended and customized on a per-application basis [COL 00]. This vision is shared with several other researchers [SIL 97, HAM 99, CHA 00, DIT 01].

Replication techniques have been heavily investigated in various areas such as group communication systems, distributed shared memories, distributed file systems, DBMS and object-oriented distributed systems. Experiments have lead to a large number of replication techniques and protocols [GRA 96, KEM 00, KIN 99, DRA 01]. Apart from the principles involved, these experiments have little in common. They generally implement some dedicated ad hoc replication support. Providing an adaptable replication support could have a major impact with the growing replication needs required to support ubiquitous computing in the large.

Adaptation can be obtained through parametrization. It seems quite impossible to provide a general purpose replication protocol that can be parameterized to accommodate all runtime environments. Indeed, we have noticed that large amounts of code need to be replaced from one case to another. Further, increasing the number of parameters tends to heavily increase the code complexity. We argue for a component-based approach where pieces of code (i.e., components) can be changed for adaptation purpose. This requires components of the framework to be clearly identified in terms of functions, defined through relevant interfaces. Like with any component model, dependencies must be fully mastered, guarantying the ability to replace components. We also advocate that the functional scope of a component should be minimal in order to enhance its reusability.

We have not found any work on replication with adaptability as the main objective. Compared with existing work (eg. Garf [GAR 95], services for Corba [MAF 95, NAR 02], Core [BRU 95] and Globe [KER 98]), we believe RS2.7 is far more adaptable and reusable. We have considered three conditions to be fulfilled in order to optimize adaptability.

The first one is the separation of concerns, which means that the functional scope of the framework is only devoted to replication. We propose two features as the foun-

\(^{1}\) http://www-lsr.imag.fr/Les.Groupes/STORM/Storm2002/English/index.html
dictions of the framework. The first feature aims at managing bindings between replicas, mainly through their life cycle aspect (i.e., creating/deleting bindings, adding/removing a replica to/from a binding). The second feature deals with coherency between replicas of a binding, application-level consistency being built by choosing the coherency model and implementation that best fits.

The second condition is the ability to adapt to various contexts such as persistent, transactional, or fault-tolerant ones. This requires independence from potential interactions with other features such as concurrency control or save point mechanism.

The third condition is the decomposition of the coherency support into a large number of components, each one playing a reduced role. It can be decomposed with respect to two dimensions: functional and scheduling. Defining these components is probably one of the main contribution of our work. This fine grain decomposition allows the tuning of the replicas coherency to be as optimal as possible. This is done simply by assembling the most optimal implementation of a given function (e.g., a policy).

This paper is organized as follows. Section 2 gives the definition and the scope of the replication framework. Section 3 presents how it takes part in various contexts. Section 4 proposes an abstract coherency protocol used in the design of the framework (scheduling decomposition) and Section 5 presents the functional architecture of the framework (functional decomposition). Section 6 briefly reports our implementation achievements whereas Section 7 is devoted to related work. Our conclusions and future work are given in Section 8.

2. A replication framework

2.1. Definition

Even if several efforts have been devoted to propose replication frameworks [GAR 95, MAF 95, NAR 02, BRU 95, KER 98], there is no consensus on the definition and functions provided by such a framework. This section introduces our proposed replication framework through the functions it covers and its positioning with respect to the applications as well as to other features.

We argue that separation of concerns is a key issue to adaptation. In that respect, we do not intend to provide a full-fledged framework in terms of replication functions, as it reduces reusability. RS2.7 focuses on two features: life cycle management of groups of replicas (e.g. their creation and deletion) and inter-replicas synchronization protocols (named coherency protocols in the following).

Note that in general, replicated object management is not reduced to the two features that RS2.7 provides. Indeed, a replication policy usually requires the definition of the following points:

– the replication time: when to create or to delete a replica inside the system?
– the replication degree: how many replicas have been or may be created?
– the replicas placement (if needed): where to place a replica among a set of distributed nodes?
– the coherency model: what is the required coherency model?

![Replication Policy
replication time
replication degree
replicas placement
coherency model]

![Replication Framework
coherecy protocol
life cycle management]

Figure 1. Replication policy / replication framework

The coherency model is supported by the replication framework (see Section 2.2) but the replication time, the replication degree and the replicas placement are related to other services (figure 1). These services may use the replication framework for different purposes such as load balancing or fault tolerance. In these later cases, the choices of replication time and degree will certainly differ.

2.2. Coherency models and protocols

RS2.7 provides a generic approach allowing particular instanciations of the framework to obtain the appropriate coherency protocols. Requirements of the system/application using the replication framework lead to different RS2.7 instances. For example, one provides a simple master-slaves protocol while another one provides a ROWA protocol. Some applications need strong coherency whereas others may accept a divergence among replicas. For this reason, we introduce the notion of coherency model, which defines how users perceive the different replicas of an object. It can be considered as a contract between the replication framework and its users. This includes the definition of both, access and synchronization events. An access event on a replica is either a read or a write operation, while a synchronization event is a request to synchronize replicas. We can classify coherency implementations (i.e., protocols) into the four models below:

1) **One copy equivalence model**: In this model, replicas are always equivalent. Reads on replicas give up-to-date data. Writes are always executed on up-to-date replicas. Synchronization is done to preserve one copy equivalence but may not be done at each write operation. Protocols like ROWA, ROWAA, quorum, active replication, passive replication, or eager replication used by DBMS implement this model.
2) **Divergent replicas model:** This model offers *weak coherency* as it allows replicas divergence. However, it is possible to characterize this divergence by giving some guarantee on R/W operations and their execution order. For example, a sequence of operations performed on a particular replica is perceived in the same order by other replicas, or operations appear after operations that logically precede them. Guarantees on operations may force the execution of writes on an up-to-date replica. In this case, writes can be lost or two replicas can be modified simultaneously.

Protocols implementing a divergent replicas model are often used in Distributed Shared Memories (DSM) as they contribute to popular consistency models (e.g. causal consistency, PRAM consistency, entry consistency, release consistency).

3) **Convergent replicas model:** This model allows replicas to diverge but they eventually converge at some point. A limited level of inconsistency based on conditions [GAL 95] like delay, periodic, time points, version, numerical, object, or event conditions is supported. Synchronization is done when conditions will be violated. It is not possible to access replicas not respecting a condition. For many applications, permitting temporary inconsistencies between replicas is not a drawback and allows better performance. We distinguish between two kinds of models:

   a) **Convergent replicas models with reads on divergent replicas:** each replicated object has an owner that stores its current value. Updates (always done on up-to-date replicas) are first applied to the owner and then propagated to other replicas. Reads may be performed on any replica. Protocols like lazy master replication, epsilon-serializability in DMBS implement this model.

   b) **Convergent replicas models with writes on divergent replicas.** Reads and writes may be done on not up-to-date replicas. Two nodes may simultaneously update their replica and race each other to install their updates at other nodes. The replication mechanism must detect this situation and must reconcile the two processes so that their updates are not lost. Protocols like multi airline reservation, or protocols used in mobile computing implement this model.

Various requirements concerning concurrency control or fault tolerance have to be considered when implementing specific coherency protocols. Thus, replication, concurrency control and fault tolerance are not independent even though separating them and clearly defining their interactions is a way to enhance adaptability.

For example, concurrent accesses on different replicas imply dependencies between replication and concurrency control. In fact, coherency protocols define some concurrency control requirements:

1) Some protocols implementing the one copy equivalence model (model 1) perform updates synchronously on all replicas whereas some others propagate updates asynchronously. In both cases there are particular concurrency control needs.

2) The divergent replicas model and the convergent replicas model with writes on divergent replicas (models 2 and model 3b) allow several simultaneous writers and readers working on different replicas.
3) The convergent replicas model with reads on divergent replicas (model 3a) authorizes one writer and several readers simultaneously on different replicas.

Thus, during a write operation, coherency protocols need to lock all replicas when implementing the first model, while implementing the third one needs locking only one replica.

Other interactions between coherency protocols and concurrency, and also with fault tolerance are further discussed in section 5.

3. Support of consistency models

The previous section introduces the scope of RS2.7 in terms of coherency models. However, application execution requires a consistency model. A consistency model is a more or less formal specification of how the memory appears to an application. Consistency models are implemented by consistency protocols that manage objects taking into account replication, concurrency control and fault tolerance. In presence of replicated data, we argue that a consistency model (protocol) includes a coherency model (resp. protocol) (figure 2).

![Figure 2. Support of consistency models](image)

Consistency models have been adopted in several domains. In DSM (distributed shared memories), they define the value to be returned by a read event during the execution of parallel programs. Many different consistency models exist ([KIN 99]): sequential, causal, PRAM, weak consistency, entry consistency, release consistency. In this context, consistency protocols take into account concurrency and replication. DBMS also proposes consistency models. For instance, a very popular correction criterion for replicated databases is one copy serializability. This correction criterion ensures that an execution on replicated data is equivalent to an execution working with a single copy of data, and that the execution of transactions is serializable. In this case,
the consistency model takes into account concurrency, fault tolerance and replication. ACID transactions offer a protocol implementing this consistency model.

With our proposed coherency models, consistency models do not have to deal with replication issues. The objective is that issues related to replication are hidden behind the coherency models. This approach promotes the use of RS2.7 in several contexts supporting transactions, fault tolerance, DSM, etc. In order to better understand this, two examples are depicted in the next two subsections.

3.1. Transactional context

With replicated data, a transactional service aims at doing a specific consistency model. To do this, it interacts with a concurrency service and RS2.7. If the consistency model is one copy serializability, the transactional service needs 1) a concurrency service which provides one writer or several readers of an object and 2) concerning replicated objects, a one copy equivalence model. There are two levels of coherence: one among the replicas of an object (specified by the coherency model) and another among objects managed by the transactional service (in order to respect a consistency model).

Let us consider a system with two databases (DB1 and DB2), where each database contains a replica of objects A and B (i.e., replicas A1 and B1 on DB1, replicas A2 and B2 on DB2). T1 is a transaction on DB1 that modifies A and B. T2 is a transaction on DB2 that modifies A. Notice that the transaction manager is not aware of replication. The execution of T1 requires the transactional service to request a lock on A from the concurrency service and to inform RS2.7 of the write operation on A. Then, RS2.7 requests a lock on objects A1 and A2. The same process applies to object B. At the same time, the transactional service of DB2 requests A for T2. Thanks to the lock set by RS2.7, T2 is denied from locking A.

There are two levels of concurrency control: one managed by the coherency protocol and a second managed by the transactional service. The awareness of replication remains inside the replication framework. If the transactional service wants to implement epsilon serializability (relaxed isolation), the approach ensures that there is no consequence on the replication framework, but only modifications to the level of concurrency managed by the transactional service.

3.2. Mobile context

In a mobile context, the transactional service needs a convergent replicas model with writes on divergent replicas (3b). Let us consider the previous example with DB1 as a mobile host. When the transaction service on DB1 locks A, the replication framework only requests to lock A1. At the same time, the transaction service on DB2 can also lock A, actually A2, to execute T2 concurrently. A merge process could then
appear later. The fact that there is a specific kind of replication is also hidden by the replication framework.

4. An abstract coherency protocol

For adaptability purpose, RS2.7 must support several coherency protocols. Nevertheless, the large variety of existing coherency protocols complexifies the design of the framework. Protocols differ mainly in the manner and the order they accomplish different phases. In order to be able to provide a generic replication framework, adaptability is introduced by providing abstractions for defining coherency protocols. These abstractions consist of five generic phases: the access, coordination, execution, validation, and response phases. The differences between protocols are characterized by the approach they use in each phase and in the order the phases are executed. This leads to a replication framework whose design is independent of any protocol. The abstract coherency protocol is the minimal shared part between all existing protocols. To introduce the five phases, let us consider a client object that interacts with a replicated object.

**Access phase:** a client object submits a request (operation) to a replicated object. Replication is said to be transparent if the client object interacts with a logical object and is not aware of the underlying physical replicas, nor how many of them exist or where they are. On the other hand, with non-transparent replication, the client object sends its request directly to one or several (possibly all) replica.

**Coordination phase:** it includes preliminary treatments to the execution of a request. If necessary, replicas coordinate with each other to synchronize the execution of the requested operation. This phase may also include coordination actions with concurrency control and/or with fault tolerance services.

**Execution phase:** The requested operation is actually executed on the replica(s).

**Validation phase:** The replicas make sure that they agree on the result of the execution. For instance, they may decide if it is necessary to undo or to redo some actions. Interactions with concurrency control and/or fault tolerance services may be required.

**Response phase:** The outcome of the executed operation is sent to the client object. Protocols show two possibilities: either the response is sent only after everything has been settled and all protocol phases have finished, or the response is sent as soon as it is available even some phases have not been completed yet.

Some protocols may skip some phases, order them in different ways, iterate over some of them, or merge them into a sequence. Let us consider four particular coherency protocols to illustrate the five phases. In these examples, a client object submits a write request.
The ROWA protocol (Read One Write All)\cite{BER 84} performs writes synchronously on all the replicas and reads on one replica. It implements a one copy equivalence model (section 2.2, model 1). In the access phase the write request is captured. In the coordination phase, it is transmitted to all the replicas, which execute it (i.e., execution phase). The concurrency control ensures that a single client object accesses the replicated object. The validation and response phases are empty.

In the ROWAA protocol (Read One, Write All Available)\cite{GOO 83}, writes are done synchronously on all available replicas and asynchronously on the others; reads are done on one replica. It also implements a one copy equivalence model but the coordination, execution and validation phases are different from ROWA. If a replica is not available during the synchronization process, there is a loop between the validation and coordination phases. When the replicas are available, they are updated asynchronously during the coordination phase.

In the lazy master/slaves replication \cite{GRA 96}, used in the DBMS context, updates are done on the master and slaves are updated asynchronously. This protocol implements a convergent replicas model with reads on divergent replicas (section 2.2, model 3a). Thus, the access, execution and response phases are executed on the master replica. The coordination, execution and validation phases are executed asynchronously in this order on the slave replicas.

In active replication \cite{POW 91}, based on the communication primitives, the access and coordination phases can be merged, as well as the validation and response phases.

We were inspired from \cite{WIE 00} which proposes a decomposition with the objective of comparing protocols used in distributed systems and database systems. Nevertheless, they focus on strong coherence. Our abstract coherency protocol considers weak coherence among replicas. Besides, we do not include concurrency control and messages ordering issues. We consider only interactions with these aspects.

For each phase, we define a component with a generic interface\superscript{2}. Thus, it can be possible to change some specific phases of a particular coherency protocol for adaptation purposes.

While this section has defined a first decomposition for the protocols (scheduling decomposition), the next one presents the components that can be used to compose each phase (functional decomposition).

5. Functional architecture of RS2.7

In order to obtain adaptability inside the framework, functional decomposition is in turn applied to coherency protocols. This means that common functionalities have been extracted from protocols. In this respect, a functional architecture distinguishing the components involved in the construction of coherency protocols is proposed. Each

\superscript{2} Due to space limitation, we do not present the interfaces in this paper.
component has an interface covering a particular function that can be implemented in several ways. Components can be used to build coherency protocols for applications with particular requirements.

The functional architecture provides four categories of components (see figure 3):

– **Kernel components** are considered as the basic level to construct simple protocols. These components mainly concern replica life-cycle, their communication and interactions with the user application.

– **Components common to all coherency models**. This category introduces components related to general synchronization issues.

– **Components dependent on the coherency model**.

– **Components dependent on the coherency protocol**. These later two categories add particular model/protocol dependent components.

The next sections present each category in detail, and illustrate the presentation with a ROWA protocol and some of its variants.

---

3. Due to space limitation, we do not present the interfaces in this paper.
5.1. Kernel components

Kernel components participate to the access, coordination, execution and response phases. No validation issues are considered at this stage. They also provide means to manage replication life cycle.

The life cycle manager component manages the life cycle of two kinds of entities: replicated objects and replicas. This means that it supports the creation/deletion of replicated objects, as well as adding/removing replicas associated with such a replicated object.

The Access Phase involves a dispatcher manager component. The dispatcher manager component captures requests submitted by client objects and forwards them to the relevant replicas.

The Coordination Phase includes the synchronization messages factory, the synchronization messages listener manager, the group membership manager and the group multicast components. The last two components interact with the communication service (that provides non-blocking point-to-point communication and may support multicast). The group membership manager is in charge of the life cycle of object groups and maintains the list of its members. It provides support for joining and leaving groups. The group multicast component provides support for sending messages to all members of a group, with various reliability and ordering guarantees. Note that this functionality may be directly provided by the communication service or requires work from the replication framework. The synchronization messages listener manager interprets messages delivered by the communication support and transfers information to the listening components.

The Execution Phase includes two components: a local replica access manager and the local replica manager. The local replica access manager offers an interface allowing R/W on a replica whereas the local replica manager encapsulates other replica management issues (e.g. loading a replica in memory). These components contribute to the genericity of our framework as they permit the replication of any kind of objects (e.g simple or composite objects, HTML pages). The local replica access manager gives an abstract representation of replicas to the framework.

The Response Phase has a single component, response manager, passing the returning value to the client object.

Let us consider a simple ROWA protocol to illustrate the construction of a protocol using the kernel components. The phases of the protocol are access, coordination and execution. When a write is requested, the access phase performs appropriate actions through the concurrency control service before transferring the request to the coordination phase. In the coordination phase a synchronization message is constructed (by the synchronization messages factory) and is sent to all the replicas (group multicast component). The arrival of this message to the different nodes starts their coordination phase. The message is read by the synchronization message listener manager and passed to the execution phase. Thus it is applied to the local replica using the local replica access component.
5.2. Components common to all coherency models

The support of coherency models requires components for the synchronization process. The level of coherency between replicas depends on the synchronization approach. The synchronization manager component is introduced for the Coordination Phase. It is based on the synchronization messages factory (kernel component) and on two other new components: the starting synchronization component and the updates log component. The starting synchronization component decides of the synchronization time. For instance, this time may be determined by the occurrence of a read or write event, or the violation of a condition. The updates log component saves information to be exchanged during the synchronization process. It may be implemented with different mechanisms such as logs, triggers, snapshots, shadow, etc.

There are no other components common to all coherency models in the other protocol phases.

Let us consider a more complex protocol than ROWA, where the synchronization process takes place when a read or a write is requested. Thus, the starting synchronization component launches the synchronization process on read or write events. The synchronization manager requests information from the updates log component in order to synchronize replicas. Depending on its implementation, this component may return the value to be installed in the replicas or a set of operations to be executed to obtain the appropriate value.

5.3. Components dependent on the coherency model

Coherency models are characterized by the replica’s role. This role may specify which replicas can be updated by external requests and those updated exclusively by the replication framework itself. Two basic settings are possible, namely, master-slaves and peer-to-peer replication. Peer-to-peer replication allows any replica to be updated upon external request, and its modifications are eventually forwarded to the other replicas. On the other hand, a master-slaves setting defines a clear distinction between the master replica, which can be explicitly modified, and the slave replicas, which are only updated by the replication framework. Thus, the roles manager component is introduced for the Access Phase.

The treatment of conflicting modifications requires the introduction of new components in the Validation Phase: the conflict detection component and the conflict resolution component. These components are used by protocols implementing the convergent replicas model with writes on divergent replicas. In such protocols, the replicated object must be able to detect conflicts and to solve them. For this purpose, current replication systems offer different conflict resolution policies (e.g. time stamp-based, priority based, additive, maximum) that can be used in the implementation of these components. Note that interaction with the concurrency control may be required.
Let us consider a protocol allowing multiple simultaneous writers on the replicated object. Thus, after the execution phase, the validation phase (conflict detection component) checks for conflicts. If a conflict appears, conflict resolution is invoked and a coordination phase is started.

5.4. Components dependent on the coherency protocol

This category introduces components particular to specific protocols. In some protocols, it is necessary to manage the call replication problem in the Access Phase: this problem arises when a replicated object A sends a request to another replicated object B. If there are n replicas of A, B may receive n copies of the request but should execute it only once. This situation can be handled in different ways, at the client side (A) or at the server side (B). The replicated messages manager component is introduced for this purpose.

The synchronization group component is introduced for the Coordination Phase. This component decides which replicas are involved in the synchronization process. For example, in epidemic protocols, updates are forwarded gradually.

The Execution Phase may also require to order operations to be executed on local replicas. This permits executions in other orders than the arrival order. The message ordering component provides support for this feature.

The Validation Phase of some protocols requires some actions related to fault tolerance. Two components are introduced for this purpose: the consensus manager and the failing replica manager components. This last component interacts with a fault-tolerant service that detects dead replicas and decides what to do until the replicas revive. This component may interact with the group membership component. The information collector component is also introduced to collect results produced by different replicas.

As an example, let us change the ROWA protocol into the ROWAA. The components of the access, coordination and execution phases remain unchanged. It is now necessary to use a validation phase with the failing replica manager. Moreover, this phase interacts with the coordination phase. If the failing replica manager detects dead replicas, it informs the synchronization manager that handles the differed synchronization process.

With the RS2.7 component approach, we have demonstrated how replication protocols can be built by assembling components. We have also shown that existing protocols can be easily enhanced by incrementally adding new components to existing assemblies. This characterizes the ability of our framework to smoothly adapt to various situations that can largely differ between each other.
6. Implementation and experimentation

This Section presents an implementation of RS2.7 (section 6.1) and an experimentation to validate our approach. The validation has two main objectives: to show the adaptability inside the framework (section 6.2) and the adaptability to the application context (section 6.3).

6.1. Implementation

The basic architecture principle that governs RS2.7 consists in interposing mediation objects to access replicas in order to provide transparency. They are then linked to binding objects that bind replicas between each other. The mediation objects also manage replication by implementing the appropriate coherency protocol.

(1) Binding b1 = bf.createBinding();
(2) b1.setReplica(p1);
(3) Name np = binder.export(b1);

(a) Memory 1

(1) Binding b2 = bf.createBinding();
(2) b2.setReplica(p2);
(3) binder.bind(np,b2);

(b) Memory 2

Figure 4. (a) Replicable object creation / (b) Replica creation

The process to obtain a replicable object is the following (figure 5a and figure 5b):

– To ask to a binding factory (bf) the creation of a new binding representative (figure 4a line 1),

– To associate the application object p1 (this is the first replica) with this binding representative (figure 4a line 2) and

– To export the binding in the domain of replicable objects (figure 4a line 3). A binder is a particular kind of naming context that can define a name for a replicable object and that can associate a binding to it. The name associated to a binding is ensured to be unique.

Figure 5. Binding in RS2.7
To create a new replica, the process is the following (figure 5c):

- To ask a binding factory (\(\mathcal{B}\)) the creation of a new binding representative (figure 4b line 1),
- To associate the application object (this is the second replica) with this binding representative (figure 4b line 2), and
- To bind this object with the relevant replicable object with name \(\mathcal{P}\) (figure 4b line 3).

The coherency protocol that manages a replicable object is distributed on each replica through the binding representatives. The construction of a particular protocol consists in assembling the appropriate functional components (Section 5) within each phase of the abstract protocol (Section 4), and in integrating these different phases (scheduling components) through the relevant scheduling algorithm. All this is embedded into binding objects.

The implementation of a simple lazy master/slaves protocol illustrates this process. In this protocol, writes are permitted only in master replicas, and reads are allowed on all replicas (slaves replicas). The master sends updates asynchronously to all the slaves. Thus, this protocol implements the convergent replicas model with reads on divergent replicas (Section 2.2 coherency model 3a).

In this implementation, the behaviour of binding representatives is similar for all slaves but differs for the master. The same phases are executed on all master and slaves (i.e., access phase, coordination phase, execution phase and response phase) but the active components inside them differ. For the master, the access phase consists of a role manager component, the coordination phase consists of a starting synchronization manager, an updates log, a synchronization messages factory and a communication manager component, the execution phase consists of an access replica component and the response phase consists of a response manager. For the slaves the coordination phase consists of a synchronization messages listener manager. The other phases are identical.

The sequence diagrams of figure 6 show the process involved inside the framework when performing writes, reads or synchronization. When a read operation is submitted to the master or to a slave (figure 6b), this operation passes first through the access phase where the role manager decides if the operation is allowed. Next, the execution phase executes the operation accessing the access replica manager. The response is sent to the caller during the response phase by the response manager.

A write operation submitted to a slave passes first through the access phase, where the role manager decides that a read is not allowed. An error message is sent by the response phase by the way of the response manager. On the other hand, the same operation submitted to the master (figure 6a) passes first through the access phase and next through the coordination phase. The operation is recorded by the updates log, next ex-
executed by the execution phase accessing the access replica manager. Asynchronously the starting synchronization manager (figure 6c) starts the synchronization process according to conditions. In this case, the coordination phase constructs the synchronization message with the synchronization message factory and the updates log. This message is sent to all replicas by the communication component. Each slave receives this message and updates its associated replica performing the execution phase (figure 6d).

We now illustrate the advantages of our approach by showing how it can be adapted to changes in the coherency protocol (section 6.2) and to application contexts (section 6.3).

6.2. Adaptability inside the framework

In order to validate the decomposition of the replication functionality proposed in sections 4 and 5, several protocols, which implement the four coherency models (Section 2.2), have been developed in Java. For the time being, the composition of these components and the composition of the phases are hand coded. Our first goal is to validate functional and scheduling decompositions.

A first kind of adaptability inside the framework is obtained by the fact that each component of the functional architecture has a generic interface, allowing various
implementations. For instance, the update log component of the previous example (Section 6.1) can be based on a log or a file or merely by directly performing the execution phase, using its access replica manager to access the replica state. In the same way, if the replica is an HTML page, only the access replica manager in the execution phase has to be changed. A dynamic master could also be introduced. In this case, the binding representatives are the same for all the replicas. The role manager decides which replica is the master according to a specific algorithm. An implementation can be an election between all replicas or a “token ring”-like algorithm. Conflict detection and conflict resolution components can be typically implemented in several ways according to the context.

A second way to obtain adaptability inside the framework is to modify a phase (adding, deleting components as well as changing the composition of the basic components). For instance, a dispatcher component can be added to the access phase of the slaves. Thus, when a write operation is submitted to a slave, this operation is forwarded to the master. As a second example, the starting synchronization manager can be suppressed, then starting the synchronization process when there is a write operation on the master. This modification allows the initial protocol to evolve into a one copy equivalence protocol (model 1 in section 2.2). In this case the coordination phase also needs to interact with a concurrency control that locks all replicas during the write operation.

**Figure 7. Read on a slave in a pull lazy master/slaves protocol**

A third kind of adaptability inside the framework consists in changing a binding representative. For instance, the slave binding representatives can synchronize with their master before a read operation (figure 7). In section 6.1, the protocol uses a push policy for synchronization, while it uses a pull/push policy here: the slaves pull updates before reads and the master pushes them according to its starting synchroniza-
tion manager. If the master binding representative is not changed, it sends updates to all replicas when a particular replica will be read. The master binding representative can be modified for sending updates only to the replica that will be read.

6.3. Adaptability to the application context

A utilization of the replication framework has been done in the context of the PING (Platform for Interactive Networked Games) European project [ENS 01]. The PING project intends to specify, develop and demonstrate a flexible and scalable architecture for large-scale interactive multi-participant applications over the Internet. The context is not transactional. In this implementation (developed in Java), the focus has been put on the separation of concerns among coherency protocols, concurrency control and consistency protocols as described in section 3. Several coherency protocols have been implemented for the convergent replicas models with reads on divergent replicas (section 2.2 model 3a) and with writes on divergent replicas (section 2.2, model 3b), but without decomposition inside the binding representatives (for performance reasons). These coherency protocols are used to implement several consistency models (sequential and several variant of causal models).

Our framework is also under validation within a transactional context, within a fault-tolerant context, and within a mobile transactional context as well. So far, no modification has been required to the framework.

7. Related Work

Replication is frequently used in group communication systems, shared memories, file systems, DBMS and object oriented platforms. A large number of ad hoc replication techniques and protocols have been proposed [GRA 96, KEM 00, KIN 99, DRA 01]. There is also much academic and industrial activity on the design and implementation of adaptable replication frameworks.

A standard approach to obtain adaptability is separation of concerns between the application code and the replication code is to use inheritance: objects inherit adequate behaviors from a set of predefined classes. Another approach is to use reflexive facilities that rely on two object levels: a base level and a meta level (Garf [GAR 95], Core [BRU 98], RepliXa [KLE 96], Globe [Van 99]).

Beside this separation of concerns, our proposal also considers separation of concerns between the replication framework and other features.

Concerning supported replication protocols, existing work may be classified into two categories: those proposing a limited support of replication protocols to achieve fault-tolerance and those proposing a more generic approach.

In the first category, the range of provided replication techniques is limited to strong coherency among replicas. Their objective is to provided fault-tolerance through
replication techniques like active replication or passive replication [LIT 94]. For example, Garf [GAR 95] is an object-oriented environment that simplifies the development of fault-tolerant applications by separating the distributed behaviors of objects from their functionality. It offers a library of ready-to-use components, behavioral objects, providing adequate support for fault tolerance through replication (active and passive replication). Behavioral classes are implemented using the Isis toolkit, which provides fault tolerance at the Unix process level. The replicas are managed using group communication, implemented through multicast primitives. These primitives must ensure that component failures do not compromise the consistency of the logical state managed by group members. We find a similar approach in work around Corba [MAF 95, NAR 01, NAR 02, FEL 00]. They combine a Corba programming environment with a group communication system like that found in Isis.

Systems of the second category allow a large number of replication techniques [BRU 98, BEE 96, KER 98]. Core [BRU 95, BRU 98] is an architecture and a runtime environment for adaptable replicated objects. A replica consists of three components. First, a local copy that contains the state and offers an interface to manipulate it. Second, an access object which wraps the local copy and controls access to it. Third, a consistency manager that cooperates with the access object to maintain the consistency of the local copy. Local copies and access objects are specific, but consistency managers are generic. One selects a replica control strategy by instantiating the appropriate consistency manager. The consistency manager mainly handles locking, synchronization and forwarding of updates. The Globe [KER 98] system also provides a flexible framework for associating consistency models with distributed objects. Some other work proposes to manage replication in an adaptable way in the DSM (Munin [CAR 91], TreadMarks [AMZ 96], Midway [BER 93] Arias [D’ 96]). These systems offer to the application developer different implementations of consistency models.

We propose a framework to implement any protocol. Thus, our proposal is more general than the first category. Moreover, in our opinion, works of the second category propose more than a replication framework. They combine concurrency, replication and consistency protocols. They do not really isolate what is specific to replication and do not extract general replication functionalities. This limits adaptability, flexibility and reusability of their frameworks.

8. Conclusions and future work

This paper contributes to a clear separation of the replication functionalities in order to enhance adaptability. The RS2.7 replication framework provides support for replica life cycle management and for coherency protocols.

It is used along with other services in order to build various replication policies. These services use the life cycle management to create and delete replicas according to the policies (when to create/delete replicas, how many, where to create them).
The replication policies are also composed by the coherency model supported by the replication framework.

A coherency model defines how users perceive the different replicas of a replicated object. Four coherency models are proposed: one copy equivalence model, divergent replicas model, convergent replicas models with reads on divergent replicas, and convergent replicas models with writes on divergent replicas. A large variety of protocols can implement these models. The coherency model is part of the consistency model. This separation between coherency model and consistency models permits the use of our framework in several contexts like transactional or fault-tolerant ones.

Moreover, the RS2.7 replication framework is intended to be adapted to support a large number of coherency protocols. Although designing such a generic layer is difficult, an abstract coherency protocol has been proposed. It distinguishes five logical phases: access, coordination, execution, validation and response phases. Flexibility to support various coherency protocols is guaranteed by a functional architecture that identifies several components involved in the construction of coherency protocols. These components also allow a better composition between the replication framework and other services like fault tolerance or concurrency control.

A first implementation has been made in order to validate functional and scheduling decompositions. A second prototype of RS2.7 has been achieved and is integrated in a Platform for Interactive Networked Games (PING [ENS 01]). This experience has focused on the separation of concerns between coherency protocols, concurrency control and consistency protocols. Performance issues are considered.

On-going work includes the use of the ObjectWeb Fractal component framework[^4] [COU 01] that allows to describe the composition and to generate an optimized implementation. This approach permits to take advantage of the openness of our framework and of its internal architecture. This composition could be static or dynamic, depending on trade-offs to be made between performance and dynamic adaptation (i.e., dynamic reconfiguration). In this respect, it is also intended to experiment component merging patterns for performance enhancement. Moreover, transaction contexts are heavily investigated with RS2.7, and especially environments like EJB platforms.

Acknowledgements

A *grand merci* to Elizabeth Perez Cortez, Patricia Serrano-Alvarado, Alexandre Lefebvre and NODS members for very useful discussions on our work.

[^4]: http://www.objectweb.org/architecture/component/index.html
9. References


