# Adaptable Replicated Objects in Distributed Environments

Georges Brun-Cottan, Mesaac Makpangou

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

# *Adaptable Replicated Objects in Distributed Environments*


Georges Brun-Cottan
Tel : +33 1 39 63 54 26
Georges.Brun-Cottan@inria.fr

Mesaac Makpangou
Tel : +33 1 39 63 52 93
Mesaac.Makpangou@inria.fr



**N° 2593**

Mai 1995

PROGRAMME 1




ISSN 0249-6399

INRIA
ROCQUENCOURT

# Adaptable Replicated Objects in Distributed Environments

Georges Brun-Cottan
Tel : +33 1 39 63 54 26
Georges.Brun-Cottan@inria.fr

Mesaac Makpangou
Tel : +33 1 39 63 52 93
Mesaac.Makpangou@inria.fr

Programme 1 — Architectures parallèles, bases de données, réseaux et systèmes distribués

Projet SOR

Rapport de recherche n° 2593 — Mai 1995 — 23 pages

**Abstract:**

This paper presents an architecture and a run-time support environment for adaptable replicated objects. The architecture separates the type-specific logic of a replicated object, e.g. concurrency control, from the generic logic, e.g. consistency management. A replicated object is structured into components with generic interfaces. A programmer can adapt a replicated object to the application specifics by replacing components that implement generic logic. Our architecture, while making the replicated object flexible, does not preclude optimizations based on object semantics or on application specifics. A tool-box of ready-to-use objects reduces development cost for new replicated object types.

**Key-words:** replication, consistency management, fine grain concurrency control, object system, distributed system

*(Résumé : tsvp)*

# Objets répliqués adaptables en environnement réparti

**Résumé :**

Ce papier présente une architecture et un environnement d'exécution pour des objets répliqués adaptables. L'architecture sépare les mécanismes dépendant du type de l'objet (ex. le contrôle de concurrence) des mécanismes génériques (ex. le contrôle de cohérence). Un objet répliqué est structuré en composants munis d'interfaces génériques. Un programmeur peut adapter un objet répliqué au besoin d'une application, en remplacant les composants génériques. Notre architecture, tout en rendant l'objet répliqué flexible, n'exclut pas les optimisations basées sur la sémantique de l'objet ou les caractéristiques applicatives. Une boîte à outils d'objets prêts à l'emploi permet de réduire les coûts de développement de nouveaux types d'objets répliqués.

**Mots-clé :** réplication, contrôle de cohérence, contrôle de concurrence à grain fin, système à objets, système réparti

# 1 Introduction

Replication is recognized of primary interest to enhance availability and reliability in distributed environments. Replicas need to be maintained consistent, which raises the issue of the *consistency contract*: which consistency semantics is appropriate for a particular application ? How is the contract implemented without jeopardizing overall performance ?

Different applications require different contracts concerning consistency of replicated state, or different tradeoffs between performance and consistency. Moreover, one particular choice might change in a different networking environment or when the application evolves.

We are aware of three approaches, proposed by other authors, to adapt the consistency contract to application specifics. First, application programmers might choose among a range of consistency management protocols supported by the system [4, 6, 7]. Second, programmers can use basic building blocks [5] to implement easily a suitable consistency management protocol. Third, programmers can implement the best consistency contract, relying on structuring models making the consistency management protocol a first class object [11, 18].

These approaches address only part of the problem. The first one focuses on performance but does limit the choice of programmers to a predefined set of protocols. The last two leave application programmers with the burden of programming their consistency contract.

This paper presents an architecture and a run-time environment for *adaptable replicated objects*. For flexibility, the proposed architecture structures a replicated object into three kinds of components: access objects, replicas and consistency managers. A *replica* encapsulates a local copy of the replicated state and offers an interface to manipulate this state. An *access object* is a wrapper that controls accesses to its associated replica. A *consistency manager* cooperates with access objects to maintain the consistency of the replicated state. Access objects and replicas are type-specific whereas consistency managers are generic. That is, the internal logic of a consistency manager does not depend on any information related to the type of the replicated object. Despite being generic, a consistency manager supports fine grain concurrency control, thanks to type-specific *intention* arguments passed by access objects. An intention is a generalized lock with a generic interface to compare it with other intentions.

To reduce the development cost of new replicated object types, the run-time environment offers BOAR, a tool-box of ready-to-use components. Good candidates for inclusion into BOAR are consistency manager classes implementing commonly used consistency contracts. Each consistency manager class implements a particular consistency contract. Access and intention classes of commonly used replicated object types (e.g. structured document, tree and container) are good candidates too. Looking to the future, our ultimate goal is to ease a component-based programming. Highly skilled (system) programmers

will develop consistency manager classes as well as access, intention and replica classes for general-purpose replicated object types. These classes are then make available to application programmers. Hence, the programmer of a new replicated object type has just to figure out which components of its replicated object type are missing from BOAR and then concentrate on the development of those components. Newly-developed components can then be added into BOAR. Once added, these components can be used by both existing and new applications.

Our work combines into a single framework the advantages of the three approaches cited above, without the drawbacks. In particular, the proposed architecture does not preclude optimizations; e.g. consistency management may exploit the semantics of the object methods for fine grain concurrency control. Also, our approach does not leave the entire programming effort to programmers; rather, over the time, the task of programmers will be continuously reduced as it is likely that most components will be found in BOAR.

The rest of the paper is organized as follows. Section 2 presents the programming model and introduces two applications that have been built upon our system. These applications are used as running examples in Section 3 to present our architecture. Section 4 then discusses some related work. Finally, Section 5 draws some conclusions.

# 2 Background

## 2.1 Programming model

We assume the following definitions in the rest of the article:

- A *client* is any entity making invocations on a replicated object. This can be a thread, a process, a transaction or even a human being. We assume that a client is aware that objects are potentially shared with other clients.
- An *activity* is a set of data accesses limited in time. For example, an activity on a document structured into sections may consist of modifying sections 1 through 3.
- An *activity domain* consists on the set of logical parts of an object accessed by an activity. For instance, the activity domain for the above activity example includes sections 1, 2 and 3.
- A *consistency contract* specifies the properties that the consistency management protocol must fulfill. Examples of such properties include atomicity, isolation, and those that underly various models of memory consistency.
- A *consistency protocol* implements a consistency contract. A consistency protocol includes the detection and the resolution of conflicts among

concurrent accesses to the replicated object, i.e. concurrency control. A consistency protocol also provides mechanisms to propagate replica updates.

An application is composed of processes. Each process identifies a separate address space, and may contain multiple threads of execution. Each process with access to a shared replicated object holds, in its address space, a replica of this replicated object. Processes communicate by sharing replicated object; replicas of an object in turn communicate via updates messages sent and received by their consistency manager.

## 2.2 Running examples

The architecture described in the rest of this paper has been used to implement two distributed applications: A distributed cooperative editor and a simulation of a parking lot. The purposes of these applications are to evidence:

- The fine-grain and operation-specific concurrency control.
- The genericity of the consistency management
- How different contracts can be usefully selected according to realistic tradeoffs. For instance, how a controlled weakening of consistency may permit a better concurrency while still ensuring correctness.

The replicated objects used by these applications are described briefly below.

### 2.2.1 Replicated Document

We have added shared replicated buffers to the well-known GNU Emacs editor. In a distributed cooperative editing session, the cooperating Emacses hold replicas of shared document.

A document contains a LaTeX text structured into sections, subsections, paragraphs, and so on. The document interface comprises: `readSection` and `writeSection` methods to read and modify section contents; `getSectionLayout`, `setSectionLayout`, `addSection` and `deleteSection` methods to manipulate the document structure. `getSectionLayout` gets the outline of the document whereas `setSectionLayout` re-orders sections. Section contents and document structure can be modified concurrently. Invocations to `addSection` (resp. `deleteSection` and `getSectionLayout`) do not conflict with one another. Invocations to `readSection` do not conflict with one another, nor with any of the three former kinds of invocations.

### 2.2.2 Replicated Counter

The parking lot simulation considers a parking lot with multiple entrances. The distributed space allocator enforces the following invariant: at any time, the number of cars in the parking lot is less than its capacity. One application component runs at each entrance. They share a replicated counter representing the number of free spaces in the lot.

The counter, represented as a replicated integer, exports methods `increment` and `testAndDecrement`. The counter internally enforces the invariant of remaining nonnegative. When a car leaves the lot, the counter is incremented (`increment`). When a car arrives, the method `testAndDecrement` is invoked: this method atomically tests if the car can enter and if so, decrements the counter; it returns true if the test succeeded, and false otherwise. Invocations to `increment` do not conflict with one another.

## 3 Architecture

Our framework must capture all accesses to replicas, in order to ensure that they remain consistent with required contract. These two key function are performed by the *access object* and the *consistency manager* (see Figure 1).

A client accesses a replicated object via its access object (AO), a wrapper encapsulating the local replica. An AO enforces the following access protocol when a client accesses the object. First, the AO acquires a lock from the consistency manager. Second, it invokes the local replica, notifying any modification to the consistency manager. Third, it releases the lock.

A client chooses a consistency contract by instantiating the appropriate consistency manager. The consistency manager implements the consistency contract, makes decisions to grant locks, and propagates updates.

The rest of the section is organized as follows. Section 3.1 discusses how to exploit, in a generic way, the object semantics to achieve typed concurrency control and the logical structure of objects to achieve fine-grain concurrency control. Section 3.2 then details the architecture components, especially the basic interfaces of the access object and the consistency manager.

### 3.1 Exploiting type-specific information

One technique to increase concurrency and performance is to exploit object semantics [12, 13, 17, 22]. Another is fine-grain concurrency control [3]. Our architecture supports both, thanks to *intentions*.

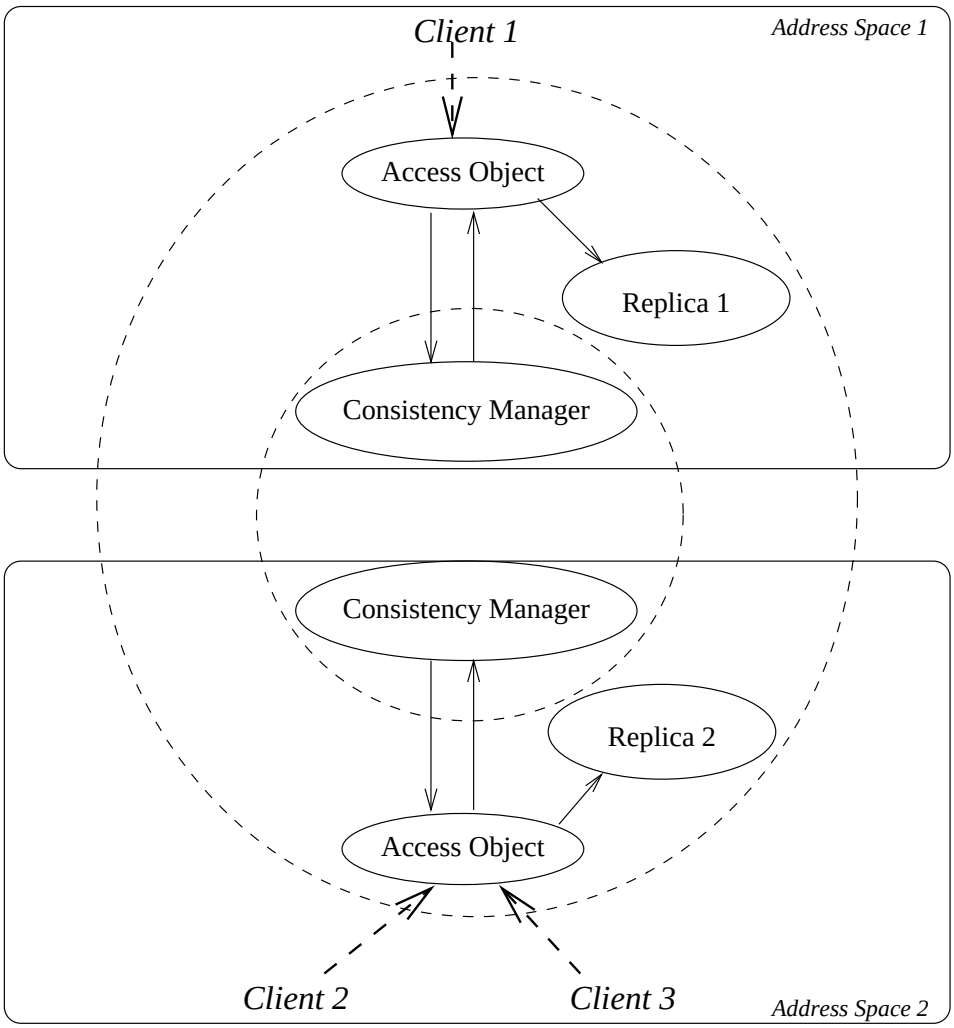


Figure 1: Concrete representation of a replicated object shared by three threads located in two processes. Dashed lines identify logical distributed object boundaries.

```
class Intention {
public:
   virtual boolean operator == (const Intention&) = 0;
   virtual boolean isWeak () = 0;
   virtual boolean isNested (const Intention&) = 0;

     // Specify which field of the intentions,
     // isConflicting should consider.
   enum dimension {DOMAIN=0x1, TYPE=0x2, CLIENT=0x4};

   virtual boolean isConflicting (const Intention&,
         dimension criterion = (DOMAIN|TYPE|CLIENT)) = 0;

   boolean test_conflict( boolean domainOverlap,
                          boolean typeConflict,
                          boolean clientCompatible,
                          dimension criterion)
};

boolean Intention::test_conflict( boolean domainOverlap,
                                  boolean typeConflict,
                                  boolean clientCompatible,
                                  dimension criterion)
{
   boolean D = criterion & DOMAIN;
   boolean T = criterion & TYPE;
   boolean C = criterion & CLIENT;

   return ((domainOverlap&&typeConflict&&clientCompatible)
                && (D  && T  && C  )) ||
          ((domainOverlap && clientCompatible)
                && (D  && !T && C  )) ||
          ((domainOverlap && typeConflict)
                && (D  && T  && !C )) ||
          ((domainOverlap)
                && (D  && !T && !C )) ||
          ((typeConflict && clientCompatible)
                && (!D && T  && C  )) ||
          ((typeConflict)
                && (!D && T  && !C )) ||
          ((clientCompatible)
                && (!D && !T && C  ));
}
```

**Figure 2:** Abstract class `Intention` defines the mandatory intention interface. `test_conflict` is invoked by derived classes of `Intention` to elaborate, according to the `criterion`, the final result of the `isConflicting` predicate.

An intention characterizes an activity (a set of data accesses limited in time) that a client plans to perform on a replicated object. More precisely, an intention identifies a client, an activity domain (i.e. the logical parts of the object that are concerned by the intended accesses) and a set of action types that this client intends to perform on the specified domain. An intention generalizes a lock.

Consider the cooperative editing application. An intention might specify some sections that the client intends to access during some activity. For each section accessed by this activity, the intention object specifies an action (e.g. contents modification and layout modification) to be performed.

Any intention class implements a basic generic interface. Figure 2 shows this interface:[1]

- `isWeak` returns true if the corresponding activity is weak. Briefly, we distinguish two categories of activities: weak and strong. *Weak activities* do not conflict with one another; that is, the order in which they are performed on different replicas is irrelevant. *Strong* activities potentially conflict with one another and with any weak activity.
- `isConflicting` returns true if its argument conflicts with the current intention along dimensions specified by the caller.

  The comparison can be done along three dimensions: the action type, the activity domain or the client identifier. In the cooperative editing example, two intentions conflict along the activity domain if their corresponding access domains overlap (e.g. they have a section in common); they conflict along the action type if the corresponding activities are not commutative; finally, they conflict along the client dimension if they are initiated by different clients.
- `isNested` returns true if its argument represents a *nested activity*. An activity $A_1$ is a nested activity of $A_2$ if the following conditions are satisfied: (i) $A_1$ and $A_2$ are initiated by the same client; (ii) the activity domain of $A_1$ is included within the one of $A_2$.

Figures 3 and 4 show the intention classes associated with the replicated counter and the replicated document.

### 3.1.1 Weak vs. strong activity

Consider the replicated counter used by our parking lot application. One can observe that: (i) to make sure that spaces will not disappear, all increments must be eventually performed on all replicas; (ii) to guarantee the space invariant, all test and decrement activities must be globally ordered. To help capture the

[1] The exact way these methods are used by the generic consistency management will be discussed in the next section.

```
// pseudo C++. Don't pay any attention to parameter
// contravariance.

enum CounterAccessType{INCREMENT, TEST_DECREMENT}

class CounterIntention : public Intention {
  CounterAccessType type;
  Client client;

public:
   CounterIntention(CounterAccessType cap, Client cl):
         type(cap), client(cl){}

   virtual boolean operator == (const CounterIntention&);
   virtual boolean isWeak ();
   virtual boolean isNested (const CounterIntention&);
   virtual boolean isConflicting (const CounterIntention&,
                                  dimension);
};


boolean CounterIntention::isWeak()
{
   return type == INCREMENT;
}

boolean CounterIntention::isNested(
        const CounterIntention& ci)
{
   return client == ci.client;
}

boolean CounterIntention::isConflicting(
                const CounterIntention& ci,
                dimension criterion)
{
  return test_conflict( 1,
                        (type != ci.type),
                        (client != ci.client),
                        criterion);
}
```

**Figure 3:** `CounterIntention` is the intention class used by the replicated counter. `Increment` operations are classified as weak, whereas `testAndDecrement` operations are classified as strong. `test_conflict` is inherited from base class `Intention`.

nature of each activity, `isWeak` should respond *true* for intentions associated with activities that only increment and *false* for others. It is then up to the consistency manager to exploit this predicate to order the overall set of activities in order to improve performance without violating the consistency contract. Figure 3 shows the interface and the implementation of the intention class used by the replicated counter.

Consider now the replicated document example. The three following alternative choices for the set of weak activities are equally valid:

- `readSection` ∪ `getSectionLayout`
- `readSection` ∪ `addSection`
- `readSection` ∪ `deleteSection`

Depending on the relative frequency of method invocations, one or another alternative will have the best performance. The current version of the cooperative editing application (see Figure 5) implements the first alternative.

### 3.1.2 "Smart Intentions"

An intention can be "smart" in order to improve the degree of concurrency. An intention can take into account the logical structure of the object and/or the object-specific set of action types. The better intentions capture object structure and semantics, the more clients can progress concurrently. To illustrate the impact of the "smartness" of intentions, consider the following two examples:

1. Suppose that two writers want to modify two separate sections of a document. Each client (more precisely, the access object operating on his behalf) creates an intention describing the intended activity.

   If intentions consider a document as unstructured, each intention will require a lock on the entire document. These intentions will conflict because they modify the same domain. Hence, in this case, two writers cannot modify concurrently different sections.

   If instead the intention takes into account the logical structure of a document (with sections, subsections and paragraphs), each intention requests a lock only for the concerned section and do not conflict. Hence, the modifications will be carried on concurrently.

2. Consider now two clients: one wants to modify the contents of some section whereas the other wants to modify the layout of the same section. These two operations do not really conflict: one modifies the document contents whereas the other modifies the meta-data describing the layout.

```
class DocumentPart {
        ... // set of document parts
public:
         // overlap (set intersection not empty)
         // on access domain
        boolean isOverlap(const DocumentPart&);

         // isNested (set inclusion) on access domain
        boolean isNested(const DocumentPart&);
}

enum AccessType { ReadSection, WriteSection,
                  GetSectionLayout, SetSectionLayout,
                  AddSection, DeleteSection};

class Access {
public :
   DocumentPart accessedPart; // access domain
   AccessType type;           // access type

   Access (const DocumentPart&, const AccessType&);

    // overlap on access domain
   boolean isOverlap(const Access&);
    // isNested on access domain
   boolean isNested(const Access&);
   boolean isTypeConflicting(const Access&);
}

class DocumentIntention : public Intention {
   Set<Access> accessedPart;
   Client client;            // client id
public:
   DocumentIntention(const Access&, const Client&)

    // we build intention incrementally.
   DocumentIntention& operator += (const Access&);

   virtual boolean operator == (const DocumentIntention&);
   virtual boolean isWeak ();
   virtual boolean isNested (const DocumentIntention&);
   virtual boolean isConflicting (const DocumentIntention&,
                                  dimension);
};
```

**Figure 4:** `DocumentIntention` class specifies the intention class for document. `Access`, `AccessType` and `DocumentPart` are specific of this implementation.

```
boolean
DocumentIntention::isWeak()
{
   return (type == GetSectionLayout) ||
          (type == ReadSection);
}

boolean
DocumentIntention::isNested(const DocumentIntention& ci)
{
   return  (client == ci.client) &&
           (∃ ? i ∈ accessedPart , j ∈ ci.accessedPart |
                 j.isNested(i) == true)
}

boolean
DocumentIntention::isConflicting(
               const DocumentIntention& ci,
               dimension criterion)
{
    return test_conflict(
            (client != ci.client),
            (∃ ? i ∈ accessedPart , j ∈ ci.accessedPart |
                 j.isOverlap(i) == true),
            (∃ ? i ∈ accessedPart , j ∈ ci.accessedPart |
                 j.isTypeConflicting(i) == true))
}
```

**Figure 5:** Details of the `DocumentIntention` class as implemented in the co-operative editor. Activities chosen as weak are those invoking the `readSection` or `getSectionLayout` methods.

If intentions consider only `read` and `write` action types, the two clients cannot perform their actions concurrently. If instead intentions capture the semantics of layout and contents, both modification will be allowed concurrently.

A key characteristic is that, despite the type-independence of consistency managers, our architecture supports semantic-specific optimizations, via smart intentions. Intentions are type-specific but consistency managers only use their generic interface to compare two intentions.

## 3.2 Components description

A replicated object is made of three kinds of components (see Figure 1): (i) replicas implementing the object semantics at each site, (ii) access objects implementing the access protocol, and (iii) a distributed consistency manager implementing the consistency protocol. The first two components are type-specific (each type of replicated object must define a specific version of these ones). The last one is generic, i.e. can be reused as is by all replicated object types.

Sections 3.2.1, 3.2.2 and 3.2.3 present these components in turn and discuss how they are related with one another.

### 3.2.1 Replicas

Data in a replicated object are encapsulated in replicas. A replica is an object located in a particular process address space. It needs not worry about concurrency or replication since other components of the architecture manage consistency transparently. A replica provides a service interface to manipulate the replicated state. We anticipate that a replica will often be an instance of a general purpose class found in widely available library (e.g. GNU libg++, LEDA, STL) unmodified.

### 3.2.2 Access objects

An access object (AO) is a wrapper encapsulating a local replica. Each AO has two interfaces : a client interface used by local clients to access the replicated object (replica) and an upcall interface used by the consistency manager to report remote modifications.

The client interface is divided in two parts: the concurrency control interface and the service interface. A client may protect a sequence of invocations under a single intention, by bracketing them between `beginActivity` and `endActivity`. The service interface is identical to that of the wrapped replica (see Figure 6). Each method of this interface implements the following access protocol:

```
class Counter {
    int *value; // the local replica.
    Consistency *manager; // the local consistency manager

public:
//** client interface **

      // concurrency control interface
    Activity& beginActivity (Intention&);
    void endActivity (Activity&);

      // service interface
    void increment (int);
    boolean testAndDecrement (int);

//** upcall interface **
    void update(UpdateBlock*); // UpdateBlock =
                               // marshaled representation
};
```

**Figure 6:** a `Counter` holds the number of free spaces in the parking lot.

1. Instantiates an intention object describing the access characteristics of the invocation and the client identifier.
2. It then invokes the consistency manager (`beginActivity`) passing the previously created intention.
3. Once the activity is allowed by the consistency manager, it invokes the corresponding method of the local replica.
4. If the methods updates data, it creates a message representing the update and sends it (via the consistency manager) to its peers.
5. Notify the end of the activity (`endActivity`) to the consistency manager.

Figures 6 and 7 show the service interface and the pseudo code for each method for the replicated counter example.

### 3.2.3 The distributed consistency manager

The *consistency manager* implements the consistency contract. This includes mainly policies for granting locks and for propagating update requests.

A consistency manager is generic. It can be used to manage efficiently the consistency of any replicated object, regardless of the replicated object type. Each consistency manager holds a reference to an AO of its associated replicated object.

```
void Counter::increment(int increment)
{
    CounterIntention intention =
          CounterIntention(INCREMENT, clientId()); // (1)

     // Ask execution permission to the consistency manager
     // It is a blocking call
    Activity activity =
            consistency->beginActivity(intention); // (2)

    *value += increment // (3) change local replica

    UpdateBlock update_block;
    marshal(increment, update_block); // function shipping

      // then pass operation to the consistency manager
    consistency->update(activity, update_block); // (4)

    consistency->endActivity(activity); // (5)
}

boolean Counter::testAndDecrement(int decrement)
{
    boolean result;
    CounterIntention intention =
        CounterIntention(TEST_DECREMENT, clientId());

    Activity activity =
               consistency->beginActivity(intention);

    if ( (*value - decrement) < 0) {
      *value -= decrement;

       UpdateBlock update_block;
       marshal(decrement, update_block);
       consistency->update(activity, update_block);
       result = true;
    } else
       result = false;

    consistency->endActivity(activity);
    return result;
}
```

**Figure 7:** The two methods of class `Counter`. The numbered steps of method `increment` are explained int the text.

Figure 8 presents the generic interface of a consistency manager class:

- `beginActivity` declares the intended actions of the new activity and asks for permission to run this activity. A call to this method blocks until the entire activity can be allowed. The method returns the identifier of the newly started activity.
- `update` notifies the consistency manager that the local replica has been modified. The first argument identifies the activity that performs the modifications; the second describes these modifications.

  With respect to update reports, the consistency manager acts as a channel between replicas: it propagates the update reports to all AOs (via their upcall interface). The protocol implemented by the channel depends on the contract.
- `endActivity` notifies the completion of an activity. Client access to the replicated object is now complete.

```
class Consistency {
public:
    virtual void update( Activity&, UpdateBlock*);
    virtual Activity& beginActivity(Intention&);
    virtual void endActivity(Activity&);
};
```

**Figure 8:** The generic interface of **Consistency** manager

### Generic Concurrency Control

Upon invocation to `beginActivity`, the consistency manager determines the group of manager that should participate to the decision of authorizing the requested activity. If the intention predicate `isWeak` is true the local consistency manager can decide alone. Otherwise, all consistency managers participate in the decision.

Each participating consistency manager first checks for local pending requests. If no request is pending and the requested intention doesn't conflict with ongoing activities (`isConflicting`), the request is granted. If any pending request exists, the request is granted if the requested intention is nested (`isNested`) in one of the ongoing activities and doesn't conflict with any of them. If neither of these two conditions is satisfied, the requested intention is queued in the list of pending requests.

In a nutshell, conflict detection and classification of activities (weak vs. strong) are delegated to type-specific entities, intentions, whereas the resolution of

conflicts and the synchronizations of accesses are implemented by a generic engine, the consistency manager.

### Examples of consistency contracts

Consider the replicated counter used in our parking lot simulator. We have identified two interesting consistency contracts for this replicated counter.

The first consistency contract guarantees that no client can be denied a space (or asked to wait) if, for sure, one is available in the lot. To enforce this contract, consistency managers cooperate to execute all strong activities (i.e. invocations to `testAndDecrement()`) in the same order on all replica. They must also guarantee that, on each replica, the same set of weak activities, i.e. invocations to `increment`, precede a particular strong activity. This contract requires a lot of synchronization. In compensation, a resource will not remain unused if some client is waiting for it.

The second contract is weaker than the first, though still guaranteeing the application invariant. Briefly, consistency managers share a token; the consistency managers that holds the token can execute strong activities (i.e. invocations to `testAndDecrement`) without any synchronization with its peers. When a strong activity is requested, a consistency manager first checks if it has the token; if it doesn't own the token, it requests it from the current owner. A consistency manager receives with a token, the current value (a global synchronization is done when passing the token) of the counter.

With this contract, a client might be asked to wait while space are available. In compensation, most invocations are performed locally and participants that do not compete for allocation (exiting entrances) do not need to synchronize with allocators (entering entrances). For instance, if the parking lot has one entrance and several exits, this contract requires no synchronization among them.

Though these contracts have been devised to respond to the specifics of the simulation of the parking lot, they have been successfully used for the cooperative editing application. For this application, the second contract tolerates the readings of obsolete contents and layout; the first contract does not allow that.

Both contracts guarantee the sequential consistency. The first one is linearizable whereas the second is not. The corresponding consistency manager classes are provided by BOAR. In addition, BOAR contains type-specific classes that have been used for the two applications.

# 4 Related Work

This section describes briefly some examples of support for flexible and/or adaptable consistency management.

## 4.1 The Munin tool-box of consistency protocols

Munin [4] is a Distributed Shared Memory (DSM) implementing a release consistency protocol [10, 15]. In Munin, a shared datum can be mapped into separate address spaces and hence replicated. Munin guarantees that all clients observe a consistent state of the same data.

Munin offers programmers several consistency protocols. These are protocols that were identified as of interest for applications [2, 9]. Programmers annotate shared data according to their access pattern. Munin then, executes the suitable consistency protocol accordingly to these annotations.

With respect to the consistency management, our work differs from Munin in two points. First, we aim to let programmers choose among a non-exhaustive list of consistency contracts. In contrast, Munin proposes the release consistency contract to its clients. This consistency contract is too strong for certain applications e.g. cooperative editing. Programmers of these applications might benefit from the support of weaker consistency contracts [1, 14, 16]. Second, Munin considers two access types: `read` and `write`. Our system proposes a generic approach that permits a consistency manager to exploit the semantics of object methods and hence, a richer set of access types.

## 4.2 The Shadows tool-box of basic building blocks

Shadows [5] provides a library of classes implementing basic abstractions commonly used in distributed applications. The available building blocks include class `Durable`, class `Shareable` and class `Lockable`, which implement persistence, sharing and concurrency control respectively.

Application classes inherit from these classes and may refine them. For example, replication (caching) is provided by a specialization of the class `Shareable`. Replicas are created as a side-effect of object migration: when a replicable object migrates to another address space, it leaves a shadow object acting as a replica. To have concurrency control, a replicated object must inherit from class `Lockable`.

The main interest of Shadows is that it offers ready-to-use objects, hence reducing the development cost. Shadows encapsulates independent properties such as persistence, shareability and concurrency into separate base classes that are inherited. This has two drawbacks. First, it is hard to implement optimizations

that require a close cooperation between independent properties whereas such a cooperation has been shown to be efficient [12]. Second it imposes a static choice of the consistency contract.

Unlike Shadows, our work focuses on making a replicated object adaptable without compromising performance (which is often the price to pay for flexibility). Consistency contract is dynamically chosen and our architecture has been devised to permit a close cooperation between consistency management and the concurrency control. To reduce the development cost, we propose, like Shadows, a tool-box of ready-to-use objects.

### 4.3 The Fragmented Object model and Subcontracts

The SOS system [21] is based on the Fragmented Object (FO) model [8, 18], a structure for distributed objects. Roughly, a FO is a set of *fragment objects*, local to an address space, connected together by a *connective object*. Fragments export the FO interface, connective objects implement the interactions between fragments. It is up to the FO programmer to decide if a fragment locally implements the service or is just a stub to a remote server fragment. Fragments could be replicas; in that case, the connective object implements the consistency management. The FO programmer is free to encapsulate any cooperation policy within the connective object [19].

A comparable model called Subcontract [11] is offered by Spring. The Spring Subcontract structures an object around the so-called object representation, a table of method entries and a Subcontract. Spring offers two replication Subcontracts: the replicon and caching Subcontracts. The replicon is the more basic: it binds each client to a replica and permits multi-invocation on replicas. The caching Subcontract is more elaborate. It implements the whole Spring caching management logic [20].

Although these models provide the necessary level of flexibility, it remains difficult for a naive programmer to build an efficient replicated object. For example, it is unclear how a connective object (or a Subcontract) can benefit from the operation semantics.

Our work builds upon the FO model. The baseline of the original work on FO was to hide clients the cooperation between fragments of a FO, while allowing the programmer of this FO to control the details of this cooperation. In this original work, FO programmers have to worry about details of the cooperation. The present work aims to leverage this concern for replicated object programmers. More precisely, we define a generic model of connective object which efficiently manages the consistency of any replicated object relying on it.

# 5 Conclusion and future work

We presented an architecture for adaptable and efficient replicated objects. The heart of this architecture is a generic consistency manager. The generic consistency manager exploits object semantics as well as the object's logical structure to optimize consistency management.

One key benefit of this work is that application programmers may really concentrate on the application specific logic. This approach makes distribution and replication almost transparent to application programmers. For a particular replicated object type, the programmer defines the type-specific classes and attaches the consistency manager that suits application needs. The consistency manager classes are provided by BOAR. The programmer may also find in BOAR some type-specific components.

Looking to the future, we have two objectives. First, we want to add new consistency managers into BOAR. Second, manual coding of AOs and intentions by application programmers is painful an error-prone. We would like to support automatic generation of these classes. This will be a little harder than generating simple stubs. In particular, for this to be efficient, we need a means to describe the semantics of each method as well as a means to specify the desired consistency contract.

# Acknowledgments

We would like to thank Marc Shapiro, Michel Ruffin, Julien Maisonneuve and Ian Piumarta for their valuable comments on the paper. This research is supported by the ESPRIT Basic Research Project No 6360 (BROADCAST). Georges Brun-Cottan is supported by a grant from the Ministère de le Recherche et de la Technologie and is also affiliated to the Laboratoire MASI of the Université Paris 6.

# References

[1] Daniel Barbará and Hector Garcia-Molina. The case for controlled inconsistency in replicated data. *IEEE Computer Society Technical Committee on Operating Systems and Application Environments Newsletter*, 4(3):8–11, 1990.

[2] John Bennett, John B. Carter, and Willy Zwaenepoel. Adaptive software cache management for distributed shared memory architectures. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 125–135, Seattle, WA (USA), May 1990.

[3] Luis-Felipe Cabrera, Allen W. Luniewski, and James W. Stamos. Fine-grained access control in a transactional object-oriented system. *Computing Systems*, 5(3):199–216, 1992.

[4] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. *Operating Systems Review*, 25(5):152–164, October 1991.

[5] S. J. Caughey, G. D. Parrington, and S. K. Shrivastava. Shadows - a flexible support system for objects in distributed systems. In *Proceedings of the Third International Workshop on Object Orientation and Operating Systems*, pages 73–82, Asheville, NC (USA), December 1993.

[6] Partha Dasgupta, Richard J. Leblanc, Jr., and William F. Appelbe. The Clouds distributed operating system: Functional description, implementation details and related work. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 2–9, S. José CA (USA), June 1988. (IEEE).

[7] Partha Dasgupta, Richard J. LeBlanc Jr., Mustaque Ahamad, and Umakishore Ramachandran. The Clouds distributed operating system. *IEEE Computer*, 24(11):34–44, November 1991.

[8] Peter Dickman and Mesaac Makpangou. A refinement of the fragmented object model. In *Proceedings of the Second International Workshop on Object Orientation and Operating Systems*, pages 230–234, Dourdan (France), October 1992. IEEE Computer Society Press.

[9] S. J. Eggers and R. H. Katz. A characterizarion of sharing in parallel programs and its application to coherency protocol evaluation. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 373–382, Honolulu (Hawai), 1988.

[10] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, Seattle, WA (USA), May 1990. ACM SIGARCH.

[11] Graham Hamilton, Michael L. Powell, and James G. Mictchell. Subcontract: A flexible base for distributed programming. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 69–79, Asheville, NC (USA), December 1993.

[12] Maurice Herlihy. Type specific replication algorithms for multiprocessor. In *In Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 70–74. IEEE, 1990.

[13] Maurice. P. Herlihy and William E. Weihl. Hybrid concurrency control for abstract data types. *Journal of Computer and System Sciences*, 43(1):25–61, August 1991.

[14] Phillip W. Hutto and Mustaque Ahamad. Weakening consistency in distributed shared memories. In IEEE, editor, *Proceedings of the 10th International Conference on Distributed Computing Systems*. IEEE, May 1990.

[15] Peter Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 13–21, Gold Coast (Australia), May 1992.

[16] Rivka Ladin, Barbara Liskov, and Liuba Shrira. Lazy replication: Exploiting the semantics of distributed services. *Operating Systems Review*, 25(1):49–55, January 1991.

[17] H.V Leong and D. Agrawal. Type specific coherence protocols for distributed shared memory. In IEEE, editor, *In Proceedings of the 12th International Conference on Distributed Computing Systems*, pages 434–441. IEEE, 1992.

[18] Mesaac Makpangou, Yvon Gourhant, Jean-Pierre Le Narzul, and Marc Shapiro. Fragmented objects for distributed abstractions. In T. L. Casavant and M. Singhal, editors, *Readings in Distributed Computing Systems*, pages 170–186. IEEE Computer Society Press, July 1994.

[19] Mesaac Makpangou, Yvon Gourhant, and Marc Shapiro. BOAR: A library of fragmented object types for distributed abstractions. In *Proceedings of the 1991 International Workshop on Object Orientation in Operating Systems*, Palo Alto, CA (USA), October 1991.

[20] Michael N. Nelson, Graham Hamilton, and Yousef A. Khalidi. Caching in an object oriented system. In *Proceedings of the 1991 International Workshop on Object Orientation in Operating Systems*, pages 95–106, Asheville, NC (USA), December 1993.

[21] Marc Shapiro, Yvon Gourhant, Sabine Habert, Laurence Mosseri, Michel Ruffin, and Céline Valot. SOS: An object-oriented operating system — assessment and perspectives. *Computing Systems*, 2(4):287–338, December 1989.

[22] William E. Weihl. Commutativity-based concurrency control for abstract data type. *IEEE Transactions on Computers*, 37(12):1488–1505, December 1988.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105,
78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399