HCC: A Concurrency Control Mechanism for Replicated Objects

Francesc D. Muñoz-Escoí, Pablo Galdámez and José M. Bernabéu-Aubán
DSIC-Univ. Politècnica de València

The HIDRA Concurrency Control (HCC) mechanism provides support for concurrency control in environments where the coordinator-cohort replication model is being used. This replication model allows the arrival of multiple invocations to different object replicas which serve locally those invocations and later make the appropriate checkpoints on the rest of replicas. A distributed concurrency control mechanism is needed in this environment. The HCC uses a service serializer object and a set of serializer agents placed in each replica node. As a result, since the HCC components are replicated, this mechanism is also fault tolerant. Each invocation received by an object replica is processed by the service serializer which knows the invocations that are currently being processed. So, this agent is able to block or allow the execution of arriving invocations according to their conflicts with the currently active ones and the concurrency specification made when the object interface was declared.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*network operating systems*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*distributed programming*; D.4.1 [**Operating Systems**]: Process Management—*concurrency,synchronization*; D.4.7 [**Operating Systems**]: Organization and Design—*distributed systems*

General Terms: Distributed systems, Concurrency control

Additional Key Words and Phrases: Distributed algorithms, object invocation, replication, high availability, fault tolerance

1. INTRODUCTION

There are a lot of mechanisms to ensure synchronization in object-oriented distributed environments. Some of them are based on synchronization primitives, like distributed locks with two phase locking [Gray 1979] or other mutual exclusion algorithms which use their own protocols [Bernabéu-Aubán and Ahamad 1989], and some others use programming languages as the tool needed to synchronize the access to shared data, providing data types whose operations are somehow synchronized (e.g., monitors [Hoare 1974], Ada protected types [U. S. Dept. of Defense 1983], ...). However these mechanisms either require a big amount of messages to find out which task may access the object or they do not have good expressive power [Bloom 1979] to allow different kinds of synchronization. The situation is worst if we consider a replicated shared resource whose access has to be synchronized. In this case, the typical solution relies either on two phase locking, which is very restrictive, on dynamic voting [Jajodia and Mutchler 1990], which implies a read-write locking mechanism, or on optimistic approaches [Herlihy 1990; Krishnakumar

This work was partially supported by the CICYT (Comisión Interministerial de Ciencia y Tecnología) under project TIC 96-0729.

and Bernstein 1994], which may lead to abortion of requests.

The HCC mechanism developed in the HIDRA [Galdámez et al. 1997] architecture synchronizes the accesses to replicated objects using an operation-based granularity. It does not use distributed locking, and needs a lower amount of messages to guarantee multiple types of synchronization strategies. HCC is based on extensions to CORBA [OMG 1998] IDL. So, it is independent from the programming language, and the programmer has to deal with synchronization features only when the interface is being defined. When the objects are being implemented no care has to be taken about synchronization.

The rest of the paper is organized as follows. Section 2 describes the HCC mechanism. This starts with the extensions needed in an IDL interface declaration and a description of the HCC components and several examples of the different synchronization strategies that can be implemented using this mechanism. Later, section 3 describes how this mechanism is used to manage some kinds of replicated objects. Section 4 shows similar synchronization techniques used in other distributed environments and finally, section 5 gives the conclusion.

2. THE HCC MECHANISM

The HIDRA Concurrency Control (HCC) mechanism is used in our HIDRA architecture to control the concurrent execution of multiple requests on a replicated object. This concurrency control mechanism has to check which object invocations are being executed now and it has to decide which of the arriving requests may proceed concurrently with the current active set. Thus, it provides control on the execution of whole operations.

To decide which operations may proceed simultaneously, an extension of the IDL language is used, providing information about which pairs of operations are mutually conflictive. Basing the concurrency on this property allows the implementation of multiple synchronization strategies, such as mutual exclusion, readers-writer policy, FCFS policy, etc.

2.1 Objectives

The design of the HCC mechanism has the following objectives:

—The mechanism may be used in distributed environments.

—Replicated objects have to be supported. Some replication models (e.g., the coordinator-cohort model [Birman et al. 1985] which is allowed in HIDRA) may receive concurrently different requests on different replicas of the object. These requests have to be correctly synchronized by our mechanism.

—The mechanism has to use a pessimistic approach. The object invocation mechanism used in HIDRA assumes that an invocation will be never aborted; this prevents the use of optimistic techniques to manage concurrency.

—The mechanism has to be fault-tolerant; i.e., the failure of part of the components needed by our mechanism has to be tolerated.

—We have to reduce the possibility of misuse of the mechanism by the programmer. So, the management of the concurrency control tasks has to be as transparent as possible to the programmer.

—The amount of communication (messages) needed to decide which task may proceed must be kept at a minimum.

We have chosen to extend the ORB used as the basis of our HIDRA architecture to include the components needed to provide the concurrency control mechanism. Thus, each operation invocation is checked to ensure that when it is allowed to proceed no other conflicting operation is being executed by any other task. As a consequence, the granularity of the HCC mechanism is a whole operation. Although distributed locks allow a more precise control, this solution is similar to the one used in the greater part of concurrent programming languages.

Additionally, the specification of the conflicts between each pair of operations can be made when the interface of a given object is being declared and all the support needed to manage the concurrency control tasks can be automatized and included in the ORB machinery.

2.2 Extensions to IDL

To develop the HCC mechanism, some extensions to the IDL grammar are needed. These extensions enlarge the optional parts of an operation declaration to include which other operations of each interface instance can be executed concurrently and which operations in different objects can not proceed simultaneously.

```
<op_dcl>   ::=   [ <op_attribute> ] <op_type_spec> <identifier> <param_dcls>
                 [ <raises_expr> ] [ <context_expr> ]
```

Fig. 1. Syntax of the original operation declaration.

The original syntax of an operation declaration in CORBA IDL is depicted in Fig. 1. The HCC mechanism assumes initially that all operations of the same interface are mutually exclusive when they are invoked on the same object. All other operations can proceed concurrently because they affect different objects which usually do not share any state. As a result, all non-extended interfaces are interpreted by the HCC as specifications of objects whose state is protected by exclusive operations. These operations can not be executed concurrently.

```
<op_dcl>      ::=   [ <op_attribute> ] <op_type_spec> <identifier> <param_dcls>
                    [ <raises_expr> ] [ <context_expr> ] [ <conc_expr> ]
                    [ <conf_expr> ]

<conc_expr>   ::=   "concurrent" "(" <scoped_name> { "," <scoped_name> }* ")"

<conf_expr>   ::=   "conflicts" "(" <scoped_name> { "," <scoped_name> }* ")"
```

Fig. 2. Syntax of the extended operation declaration.

However the extensions made on the IDL grammar try to overcome this limitation. The new syntax for an operation declaration appears in Fig. 2.

The `concurrent` expression gives the list of operations (that by default are in conflict with the operation being declared now) which can proceed simultaneously with this operation. All these operations have to be already declared when the `concurrent` expression appears, thus the interface compiler can check for the consistency of that interface declaration. So, if two operations of the same interface are concurrent, the declaration of this compatibility can be found in the prototype of the second operation, but never in the first one. Thus, we save an additional `concurrent` clause in the first operation prototype. However, both operations are considered mutually compatible; i.e., if an invocation of any of them is being executed, an invocation of the other one is always allowed to proceed, independently of their arrival order.

In the `concurrent` list the programmer includes the operations of the same interface which may proceed simultaneously, either because they access disjoint subsets of the object's state or because neither of them modifies the same part of that state.

IDL allows interface inheritance. The HCC considers that all operations of all the interfaces of an object can not be executed concurrently. So, to build the list of concurrent operations we need scoped names because we have to identify the interface which provides the concurrent operation (it may be any ancestor interface in the hierarchy of interfaces provided by the object).

The `conflicts` expression gives the list of operations (that by default are allowed to proceed concurrently) which now are in conflict with this operation. In this case, it is assumed that the operations in conflict are provided by two different objects, but these objects share some state and these operations access this shared state. Again, we need scoped names to identify correctly the operations in conflict.

These two clauses give the information needed to know which invocations may be allowed to proceed concurrently, even when a replicated object is being considered.

2.3 HCC Components

The HCC mechanism relies on some components that maintain and manage the information needed to allow or suspend invocations on the objects to be controlled. These components are:

—Serializer object. This object is created when a service (a group of inter-related objects) is registered in the system and it has to decide which invocations on the replicas of the objects that compose that service may proceed. To this end, the serializer receives the information about which operations are mutually conflictive and which others may proceed concurrently. This information is maintained in a *concurrency control specification* (CCS) object, that the serializer may query to find out which operations are incompatible.

As the requests arrive to the object replicas, the ORB invokes the serializer providing information about which object instance is being invoked, which operation and which invocation identifier is being used. The serializer checks if the incoming invocation conflicts with any one of the active invocations and, if so, blocks this one. As a result, each serializer has to maintain the identifiers of a collection of active (and still non-terminated) invocations and also, the identifiers and execution threads of all blocked invocations. These constitute the dynamic state of the serializer.

—ORB machinery on the server side. Before an invocation reaches the actual object it is calling, the ORB components placed on the server side have to identify and call the appropriate serializer object. The serializer blocks the invocation until no other active conflicting invocation exists in any object replica. So, when the call to the serializer returns, the ORB machinery can invoke the appropriate object replica.

2.4 Serialization of Requests

The HCC mechanism is needed in HIDRA to serialize all requests that arrive to replicas of an object that uses the coordinator/cohort replication model. In this replication model, an invocation is initially served by only one replica, which processes the request and makes at least one checkpoint to transfer the state updates to the other object replicas. Each object invocation may be served by a different replica. So, multiple invocations may be executed concurrently in all replicas of the object and some distributed concurrency control mechanism is needed.

The HCC is managed by the HIDRA's ORB components, becoming a transparent service for the application programmer. Only a requirement is made to the programmer of replicated objects: he or she has to specify in the interface declaration which operations are incompatible, as we have described in Sect. 2.2. The interface compiler generates the CCS object that has to be provided when a replicated service is registered in a running HIDRA system. As a result of this registration, the service serializer is created and it receives the CCS object that it needs to make the concurrency control decisions. Moreover, when a new replica for this service is created in a new domain or node, the ORB components in that node get the information needed to invoke the service serializer.

```
interface ServiceSerializer { // PIDL
    struct InvoCtxt {
        RoiID           Identifier,
        CORBA::TypeId   Interface,
        ObjectId        ObjId,
        Long            Operation
    };

    void Serialize( in RoiID            InvoID,
                    in TObj             TerminationObject,
                    in CORBA::TypeId    InterID,
                    in ObjectID         ObjID,
                    in Long             OperationNumber,
                    out sequence<InvoCtxt> PrecedentSet );
};
```

Fig. 3. Interface of the service serializer object.

The serialization of a request is made when that invocation arrives to the domain where the replica of the invoked object resides. The ORB components involved in the object invocation already know that it is a replicated object and, before the invocation arrives to the target object, a call is made to the `Serialize()` operation

of the service serializer. The declaration of this operation is given in Fig. 3. The arguments needed by this service serializer operation are the following:

—`InvoID`. A reference to a `RoiID` object [Muñoz-Escoí et al. 1998] that identifies the current invocation being serialized.

—`TerminationObject`. An object needed to detect when this invocation has terminated in all object replicas. See [Muñoz-Escoí et al. 1998] for details on this object.

—`InterID`. This value identifies the interface that is being invoked.

—`ObjID`. This value is internal to the ORB and identifies the specific instance that is being invoked.

—`OperationNumber`. The operation number that is being invoked in the interface `InterID`.

The serializer does not reply this invocation until it knows that the operation that must be finally invoked by the calling thread does not conflict with any other active invocation on the same replicated object. To this end, it maintains a list of currently active invocations and another list of already blocked ones (these last ones are still not active). Three of the arguments passed in a serialization request are needed to identify the possible conflicts with other previous ROIs. They are the `InterID`, the `ObjID`, and the operation number. These three arguments constitute the *invocation context* (`InvoCtxt`, see Fig. 3 for details).

```
interface CCS { // PIDL
    exception UnknownInterface {};
    exception BadOperationNumber {};

    boolean CanBeConcurrent(
        in CORBA::TypeId    FirstInterface,
        in ObjectId         FirstObject,
        in Long             FirstOperation,
        in CORBA::TypeId    SecondInterface,
        in ObjectId         SecondObject,
        in Long             SecondOperation
    ) raises (UnknownInterface, BadOperationNumber);

};
```

Fig. 4. Interface of the CCS objects.

Once the call to the `Serialize()` operation arrives to the service serializer, it follows these steps:

(1) All invocation contexts in the active and blocked lists are inspected and a call to the `CanBeConcurrent()` operation of the CCS object is made (See Fig. 4) to test if the current invocation and the inspected one can proceed at the same time.

(2) In case that the two tested operations could not be concurrent, the identifier of the operation (its `RoiID` reference) in the active or blocked lists is inserted in a set of *precedent operations* associated to the current one.

(3) When the two lists have been scanned, if the precedent operations set is empty, this operation is inserted in the list of active operations and its `Serialize()` invocation is replied. However, if the precedent operations set is not empty, the operation is inserted in the list of blocked operations. It will remain there until all the operations in its precedent set have been terminated. When this happens, the invocation context is moved to the active list and the `Serialize()` invocation is also replied.

The service serializer uses the `TerminationObject` associated to each invocation to find out when that invocation has been finished. That happens when this object receives the unreferenced notification, as it is described in [Muñoz-Escoí et al. 1998]. In this case, its `RoiID` is removed from all precedent sets where it can be found.

We can see that this synchronization support is used by internal components of the HIDRA's ORB. As a result, the HCC mechanism can not be easily used in other environments; in fact, it is only intended for being used in HIDRA. The main advantages of this mechanism are its efficiency and fault tolerance, as described in the following sections.

2.5 Expressive Power

In [McHale 1994], the *expressive power* is defined as the ability of a synchronization mechanism to implement a range of synchronization policies. So, the wider the range of synchronization policies a mechanism can implement, the greater its expressive power will be.

Bloom [Bloom 1979] gave some criteria to identify the expressive power of a particular synchronization mechanism. He proposed that six different types of information are necessary to give a good expressive power. This types are: the name of the invoked operation, the relative arrival type of invocations, the invocation parameters, the synchronization state of the resource being synchronized, the local state of that resource and history information about invocations already terminated.

The HCC is able to manage four of these six available types of information. It uses the names (in our case they are given by the `TypeId` of the interface and the operation number) of the operations being invoked, the relative arrival type of invocations (as they are received, the precedent set is built and thus, the relative arrival time is maintained), the synchronization state (because the HCC maintains which invocations are already active and which others are already serialized but they still have not been started) and it is also able to maintain the history of past invocations on each replicated object.

With all that information, the HCC can implement different synchronization policies very easily. For instance, two of the most common synchronization policies are mutual exclusion and readers/writer. To implement mutual exclusion no special action has to be taken in HCC, because it is the default policy. So, for the interface given in Fig. 5, all operations are considered mutually exclusive and their invocations are serialized in FCFS order. Other serialization orders are also possible. For instance, a priority order could be implemented if each invocation had an associated priority value and the precedent sets were built following that criterion.

```
interface BoundedBuffer {
    void    InsertItem( in Item TheItem );
    void    ReplaceItem( in Item OldItem, in Item NewItem );
    Item    GetItem( void );
    void    PrintBuffer( void );
    Item    ListItem( in Long Position );
    void    PrintItems( in Long First, in Long Last );
};
```

Fig. 5. Example of interface declaration with mutual exclusion policy.

However, the first three operations modify the state of the buffer, while the other three only read this state (to print the buffer contents, get a copy of a given item or print a range of items, respectively). So, we can modify the previous declaration to enforce a readers/writer policy. The resulting declaration is shown in Fig 6. In this case, the operations `InsertItem()`, `ReplaceItem()` and `GetItem()` cannot be executed concurrently with any other operation of the same interface because they modify the state of the bounded buffer. On the other hand, `PrintBuffer()`, `ListItem()` and `PrintItem()` only read the state of the object and all of them can be executed concurrently.

```
interface BoundedBuffer {
    void    InsertItem( in Item TheItem );
    void    ReplaceItem( in Item OldItem, in Item NewItem );
    Item    GetItem( void );
    void    PrintBuffer( void );
    Item    ListItem( in Long Position )
            concurrent( BoundedBuffer::PrintBuffer );
    void    PrintItems( in Long First, in Long Last )
            concurrent( BoundedBuffer::PrintBuffer,
                        BoundedBuffer::ListItem );
};
```

Fig. 6. Example of interface declaration with readers/writer policy.

3. HCC AND ROI MECHANISMS

In [Muñoz-Escoí et al. 1998], the HIDRA's *reliable object invocation* (ROI) mechanism is described. This mechanism is needed to invoke objects that use either the coordinator-cohort or the passive replication models. Besides giving the synchronization support already described in the last section, the HCC is assumed to be fault tolerant. To this end, the *service serializer* (SS) object is also replicated in a way that allows its state recovery in case of failures. This is achieved using *service serializer agents*(SSAs). These components are described in the following sections, where the serialization procedure is revisited and a failure analysis is also given.

3.1 Service Serializer Agents

The SSAs are the representatives of the SS object in the kernel domains of the nodes where at least one replica of that service exists. They maintain a reference to the actual SS object and invoke it when a serialization request is made.

```
interface ServiceSerializerAgent { // PIDL
    void Serialize( in RoiID            InvoID,
                    in TObj             TerminationObject,
                    in CORBA::TypeId    InterID,
                    in ObjectID         ObjID,
                    in Long             OperationNumber);

    void Initiated( in RoiID            InvoID,
                    in TObj             TerminationObject );

    void LocallyCompleted(
                    in RoiID            InvoID );

    void Terminated(in RoiID            InvoID );
};
```

Fig. 7. Interface of the service serializer agents.

The SSAs have the interface shown in Fig. 7 and they are locally invoked when a serialization is needed by any domain of their nodes. Each SSA has to maintain a copy of the invocation context for all the ROIs that have been serialized using it (this occurs when the coordinator replica of the invoked object is placed in the SSA's node) and also the precedent sets for these ROIs and the invocation contexts of all the ROIs that are members of these precedent sets (this information is provided by the SS). All SSAs maintain a copy of the CCS, although they do not use it unless they are promoted to the SS class.

The `Initiated()` and `LocallyCompleted()` operations are needed by the SSAs that are placed on the cohort domains of a ROI. The `Initiated()` operation is invoked by the ORB when it receives the first checkpoint of a ROI. It registers the `TObj` reference for this ROI. This reference is needed in case of a failure of the SS to find out which ROIs were active. The new SS can use this reference to build a new replica for the `TObj` object.

The `LocallyCompleted()` operation is used in the cohort domains to remove the `TObj` reference received in the `Initiated()` operation, enabling thus the arrival of unreferenced notifications to the `TObj` replicas. This operation is invoked when the last checkpoint in the ROI has been processed by the cohort domain.

The `Terminated()` operation is used by the replica domains of a ROI to remove the ROI from the precedent operations sets where it was present. It is invoked when the `TObj` objects have received the unreferenced notification, meaning that the ROI has been completed in all replica domains.

3.2 Serialization Revisited

If the service replicas have a local SSA, the serialization procedure is a bit different. In this case, the `Serialize()` method of the SS returns immediately and it provides in an output argument the list of invocations that form the set of precedent ROIs. Each ROI in this list consists of a RoiID reference and its invocation context. As a result, the SSA gets this list of precedent ROIs for the serialized request. Moreover, each time a ROI is terminated (this fact is detected by the TObj when it receives an unreferenced notification), all its cohort replicas invoke the `Terminated()` method to notify to their local SSAs that this ROI has been terminated and it can be removed from the precedent sets of the blocked ROIs.

So, as a result, the calls involved in a serialization request are the following:

(1) The ORB code in the chosen coordinator replica invokes `Serialize()` on its local SSA.

(2) The local SSA calls the SS's `Serialize()` operation and receives the list of precedent ROIs. As shown in Fig. 3, this operation provides this information in the `PrecedentSet` argument (this argument is not present in the SSA's interface previously invoked by the server coordinator).

(3) If the local SSA receives an empty precedent set, it replies the `Serialize()` request, enabling the execution of the controlled operation.
Otherwise, it blocks the reply until this precedent set becomes empty.

(4) When the ROI is allowed to proceed, its coordinator replica starts the operation and it eventually makes the first checkpoint. The cohort replicas, when they receive this checkpoint invoke the `Initiated()` operation in their local SSAs. This operation enables the recovery of the SS state in case of failure.

(5) When the ROI is being terminated, the coordinator replica sends the last checkpoint. The cohort replicas, once they have processed this checkpoint, invoke the `LocallyCompleted()` operation on their local SSAs, removing the `TObj` reference.

(6) Each time a ROI has made all its checkpoints and has returned a result to its client, the invoked replicas receive an unreferenced notification in their `TObj` and find out that the ROI has terminated.
When that happens, all replicas locally invoke the SSA's `Terminated()` operation to notify that the ROI has been completed.
Additionally, the SS also maintains a `TObj` replica. The SS uses this replica to find out when the ROI has terminated, updating its data structures accordingly when this happens. This implies that the ROI is moved from the active to the terminated list and it is removed from all the precedent sets.

(7) When the SSA receives the `Terminated()` call, searches the incoming `RoiID` reference in all the precedent sets, removing it. If any precedent set becomes empty, the reply for its associated serialization request is given and the ROI is moved from the blocked to the active list.

3.3 Fault Tolerance

To ensure that the HCC mechanism is fault tolerant, we have to check that several failure cases can be overcome by HCC, rebuilding the state of the SS. Two failure

cases are considered here: the failure of a given SSA and the failure of the SS and several SSAs.

3.3.1 *Failure of a SSA.* When a SSA crashes, the whole node where it resides has crashed, because our ORB support is assumed to be in the kernel domain. So, all coordinator replicas for the ROIs being controlled by this SSA have also crashed.

We need to replace the faulty SSA because it controls the activation of the blocked ROIs when its precedent operations set becomes empty. To this end, we have to describe how a ROI is restarted when its coordinator replica has crashed.

If the ROI still remained blocked, no special action has to be taken. If the client that initiated the ROI is still alive, it will reinitiate the invocation on another replica. Since the `RoiID` is still maintained by the client, the new attempt is identified as a replay by the SS and an updated precedent operations set is returned to the new chosen coordinator's SSA.

If the ROI was already active, our ORB support will choose another coordinator replica and no serialization request is initiated to do so. When the client reinitiates the invocation on another coordinator replica to pick the results of the previous attempt, its serialization request will be replied immediately by the SS as in the case described in the previous paragraph.

3.3.2 *Failure of the SS and some SSAs.* As previously shown, when this failure happens no special action has to be taken to rebuild the state of the crashed SSAs because the ROI mechanism chooses another coordinator replica for all ROIs involved in the crash.

However, the dynamic state of the SS has to be rebuilt. This state consists of two lists of ROIs:

—*Active list.* In this list we can find all the ROIs that are currently active. For each of these ROIs, the SS maintains its RoiID, its associated `TObj` replica and its `InvoCtxt` needed to find out conflicts with the arriving ROIs.

—*Blocked list.* In this list the SS maintains all ROIs that have been blocked due to either a conflict with any active ROI or a conflict with other blocked ROIs which arrived previously.
The SS adds for each of these ROIs the set of precedent operations, identified by their `RoiIDs`.

Both lists have to be rebuilt using the information maintained by the surviving SSAs. In these SSAs, we can find:

—All the information about the blocked ROIs whose coordinator replica is placed on the same node. This includes the `RoiID` and `InvoCtxts` of all the precedent operations in each precedent set and the `RoiID`, `InvoCtxt` and `TObj` of the blocked ROI.

—The `RoiID` and `TObj` references for the currently active ROIs that have made at least one checkpoint and still have not made the last checkpoint.
Moreover, the SSAs can maintain information about the active ROIs that have invoked the `LocallyCompleted()` operation but still have not invoked the `Terminated()` one.

Thus when the SS has crashed, one of the remaining SSAs is promoted to the SS class. To rebuild its active list, the following steps are taken in the reconfiguration phase of the cluster:

(1) All surviving SSAs are queried and each of them returns a list with all `RoiIDs` that have an associated `TObj` reference (due to a previous invocation of the `Initiated()` operation).
For each one of these ROIs the SSAs return its `RoiID`, its `TObj` reference and (if it can be locally found) its `InvoCtxt`. Note that a copy of the `InvoCtxt` is stored in the SSA that made its serialization request and in all the SSAs that maintain ROIs that have been serialized as its successors (i.e., that have this ROI in its precedent set).

(2) All the `RoiIDs` returned in the previous step are inserted in the active list and a `TObj` object replica is regenerated from its reference and it is associated to its `RoiID` and `InvoCtxt`.

(3) If no `InvoCtxt` could be found for a given ROI (because its coordinator node has crashed and no successor operation exists), the `RoiID` provides a `GetInvoCtxt()` operation which could be invoked to get a copy of these data.

To rebuild the blocked list, this sequence of steps is needed:

(1) All surviving SSAs are queried and each of them returns all their blocked ROIs and the precedent set for each of these ROIs.

(2) All these blocked lists are merged to build the blocked list of the new SS. Thus, the precedent sets for a given ROI are compared and the resulting precedent set only has the ROIs that could be found in all the merged precedent sets (we assume that if in any precedent set a given ROI is missing, then this ROI was detected as terminated by that SSA, which removed it from that precedent set).
For all the SSAs that have a `RoiID` in their precedent sets that other SSAs have detected as terminated, the SS does not need to invoke the `Terminated()` method. The associated `TObj` replicas of those machines still had not received the unreferenced notification, but eventually they will receive it and will invoke that operation.

(3) Finally, all precedent sets are checked to find out if they have some ROI that does not appear in the active nor in the blocked list. If that happens, that ROI is removed from the precedent sets because it corresponds to a ROI that was active but still did not make any checkpoint and whose coordinator replica crashed. A ROI of this class has to be reinitiated. As a consequence, it has to be serialized again.
When some ROI of this kind is found, the new SS also has to invoke the `Terminated()` method of all the SSAs using its associated `RoiID` as input argument.

Once these two protocols have been executed, the new SS has a dynamic state that allows the service of new serialization requests of the HCC.

4. RELATED WORK

HIDRA needs a pessimistic concurrency control mechanism to synchronize the access to replicated objects that follow the coordinator-cohort replication model. An object-oriented programming model is assumed, which may be used to build a synchronization mechanism based on the control of the concurrency with an operation granularity, as the HCC presented here.

Other concurrency control mechanisms for replicated objects exist, but the greater part of them are used in replicated databases and are based on *quorum consensus* [Kumar and Segev 1993]. These replication models assign a vote to each replica and divide the operations in only two categories (read and write). Each time a read operation must be made, the operation has to access a *read quorum* number of replicas. Also, when a write operation is attempted, it has to access a *write quorum* number of replicas. The basic property that has to be accomplished by all these algorithms is that the sum of these two quorums must exceed the total sum of all possible votes and that the write quorum must be strictly greater than half the sum of all votes. Each replica can only issue a vote each time it has to be accessed. The operations are allowed to proceed if they have collected the required vote quorum.

To deal with failures, these mechanisms use a dynamic reassignment of quorums. There are several variants of this technique, as the *available copies* [Bernstein and Goodman 1984] and the *dynamic voting* [Jajodia and Mutchler 1990] methods.

More advanced techniques are discussed in [Herlihy 1990] where two approaches are described: *conflict-based* and *state-based* validation. In the first case, operations are allowed to proceed concurrently if they commute; i.e., if they do not conflict. This is an approach equivalent to ours. The state-based validation needs to know which parts of the state are affected by each invocation. In this case, the concurrency control mechanism needs to know the value of the arguments of each invocation and the current state of each object being invoked. Although these technique allows even greater concurrency than the conflict-based one, the amount of information that needs to be managed and the access to the object state make it infeasible in our environment. Moreover, both techniques are described in their optimistic variants which can not be used in HIDRA, because we offer a model where the invocations are never aborted.

Finally, the replication model also affects the concurrency control mechanism. In a passive replication model, where only a primary replica exists, no distributed concurrency control mechanism is needed because all invocations are received by the same replica, which can use a simple local mechanism as some kind of locks or semaphores. In the active replication model all replicas receive the same invocations in the same order. As a result, a local concurrency control mechanism is again sufficient. However, the coordinator-cohort replication model is not so easy. A concurrency control mechanism for this replication model was already given in [Birman et al. 1984]. It is based on controlling the *data dependencies* and *precedence dependencies* between the operations being requested. Thus, a data dependency exists between two operations when one of them requires the result of the other before it can be started. On the other hand, precedence dependencies exist between two operations if they conflict, i.e., they access the same part of the state and at least one of them modifies that state. Although precedence dependencies are already

controlled by HCC, data dependencies need some control on the arguments of the operations. This enlarges the amount of information that must be managed by the concurrency control mechanism and does not improve so much the concurrency.

5. CONCLUSIONS

The HCC mechanism provides an easy-to-use concurrency control support for the programmer of replicated objects. The programmer only has to worry about concurrency when the interface of the replicated objects is being declared; all other support is transparently provided by HCC.

The concurrency control is given at an operation granularity and it allows the implementation of multiple concurrency control policies. Additionally, the objects involved in the HCC support are fault-tolerant, giving as result an appropriate concurrency control mechanism for the coordinator-cohort replication model of HIDRA.

Although other concurrency control mechanisms may be found for replicated object management, HCC is either more comfortable for the programmer or requires less message interchange among the agents involved in that concurrency control or provides support for a greater number of synchronization policies.

REFERENCES

BERNABÉU-AUBÁN, J. M. AND AHAMAD, M. 1989. Applying a path-compression technique to obtain an efficient distributed mutual exclusion algorithm. In J. C. BERMOND AND M. RAYNAL Eds., *3rd International Workshop on Distributed Algorithms, Nice, France*, LNCS (sep 1989), pp. 33–44. Springer-Verlag.

BERNSTEIN, P. A. AND GOODMAN, N. 1984. An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Trans. on Database Sys. 9*, 4 (Dec.), 596–615.

BIRMAN, K. P., JOSEPH, T., AND RAEUCHLE, T. 1984. Concurrency control in resilient objects. Technical report (July), TR 84-622, Dept. of Computer Science, Cornell Univ., Ithaca, NY.

BIRMAN, K. P., JOSEPH, T., RAEUCHLE, T., AND EL ABBADI, A. 1985. Implementing fault-tolerant distributed objects. *IEEE Trans. on SW Eng. 11*, 6 (June), 502–508.

BLOOM, T. 1979. Evaluating synchronisation mechanisms. In *7th International ACM Symposium on Operating System Principles* (1979), pp. 24–32.

GALDÁMEZ, P., MUÑOZ-ESCOÍ, F. D., AND BERNABÉU-AUBÁN, J. M. 1997. High availability support in CORBA environments. In F. PLÁŠIL AND K. G. JEFFERY Eds., *24th Seminar on Current Trends in Theory and Practice of Informatics, Milovy, Czech Republic*, Volume 1338 of *LNCS* (Nov. 1997), pp. 407–414. Springer Verlag.

GRAY, J. 1979. Notes on database operating systems. In R. BAYER, R. GRAHAM, AND G. SEEGMULLER Eds., *Operating Systems: An Advanced Course*. Springer-Verlag.

HERLIHY, M. 1990. Apologizing versus asking permission: Optimistic concurrency control for abstract data types. *ACM Trans. on Database Sys. 15*, 1 (March), 96–124.

HOARE, C. A. R. 1974. Monitors: An operating systems structuring concept. *Communications of the ACM 17*, 10 (oct), 549–557.

JAJODIA, S. AND MUTCHLER, D. 1990. Dynamic voting algorithms for maintaining the consistency of a replicated database. *ACM Trans. on Database Sys. 15*, 2 (June), 230–280.

KRISHNAKUMAR, N. AND BERNSTEIN, A. J. 1994. Bounded ignorance: A technique for increasing concurrency in a replicated system. *ACM Trans. on Database Sys. 19*, 4 (Dec.), 586–625.

KUMAR, A. AND SEGEV, A. 1993. Cost and availability tradeoffs in replicated data concurrency control. *ACM Trans. on Database Sys. 18*, 1 (March), 102–131.

McHale, C. 1994. *Synchronisation in Concurrent, Object-oriented Languages: Expressive Power, Genericity and Inheritance.* Ph. D. thesis, Department of Computer Science, Trinity College, Dublin, Ireland.

Muñoz-Escoí, F. D., Galdámez, P., and Bernabéu-Aubán, J. M. 1998. Reliable object invocation in HIDRA. Technical report (mar), DSIC-II/9/98, Univ. Politècnica de València, Spain.

OMG. 1998. *The Common Object Request Broker: Architecture and Specification.* Object Management Group. Revision 2.2.

U. S. Dept. of Defense. 1983. Reference manual of the Ada programming language. Technical report, ANSI/MIL-STD-1815A, DoD, Washington, D.C.