ROI: An Invocation Mechanism for Replicated Objects

F. D. Muñoz-Escoí P. Galdámez J. M. Bernabéu-Aubán
Inst. Tecnológico de Informática, Univ. Politécnica de Valencia, Spain
fmunyoz@iti.upv.es pgaldam@iti.upv.es josep@iti.upv.es

Abstract

The reliable object invocation mechanism provided by HIDRA for the coordinator-cohort and the passive replication models offers support to ensure that all the replicas of the object being invoked are correctly updated before such an invocation is terminated. This mechanism also ensures that if a primary or coordinator replica crashes, the client is able to reconnect to the previously initiated invocations, collecting their results without requiring their reexecution. All this support is provided transparently to the client of the replicated objects, which does not notice any difference respect to the invocations made to non-replicated objects. Moreover, the protocols described in the paper deal also with the failure of any of the objects involved in this kind of invocations.

1. Introduction

Several approaches have been taken in different toolkits and distributed operating systems to support fault tolerant objects. Some of them are based on checkpointing, either on stable storage or in object replicas, and some others on replicated objects. In any case, a system of this kind needs an invocation mechanism to request the services provided by these fault tolerant objects.

When replicated objects were considered, one of the first proposed approaches consisted in making replicated procedure calls [4] to invoke simultaneously all object replicas. However, this solution had to be used carefully, because inconsistent states of the replicas could be reached if different invocations were not received in the same order in all object replicas. This led to the adoption of totally or causally ordered multicast protocols to ensure the consistency of the object replicas. The use of this solution does not seem to be very attractive, because these protocols require additional messages to guarantee the correct order and this introduces delays in the requests service. Moreover, the *active replication model* [12] assumed in these cases forces all object replicas to locally serve each request, each of them executing the same actions in their respective domains. This model clearly leads to a significant waste of computing power.

Other solutions use different types of transactions to ensure that a replicated object invocation modifies the state of all object replicas and that no other conflicting invocation may see the updates until this one terminates successfully. Since replicated objects may use the services provided by other replicated objects, nested transactions [10] are needed. However, the support needed by an unrestricted model of nested transactions is expensive because it implies the use of a distributed concurrency control mechanism, a deadlock detection protocol and a deadlock resolution mechanism; besides the support needed to roll back transactions.

In HIDRA [5], we use a *reliable object invocation* (ROI) mechanism to invoke replicated objects that follow the coordinator-cohort model [2]. Besides guaranteeing consistency and forward progress of all invocations, our ROI mechanism also ensures that the invocation is completed even if some replicas fail while it is being performed. It also guarantees that in a chain of nested ROIs a reconnection is made when some server replica crashes. This reconnection is able to collect the saved results of the previous attempt, and thus, it does not need the reinitiation of that ROI.

The rest of the paper is structured as follows. Section 2 describes the problems that have to be solved by the ROI mechanism and outlines the support given by other HIDRA components to build it. Section 3 presents the agents involved in a ROI. Section 4 describes the protocol and Section 5 analyzes the most important failure cases. Finally, Section 6 compares our solution with those of other systems and Section 7 concludes the paper.

2. Reliable Object Invocation Problem

HIDRA uses object replication as a means to provide highly available objects. Different object replication models are currently supported. The ROI mechanism is used in the

models that only use an active server replica, propagating the updates to the rest of replicas using state checkpoints.

In other distributed operating systems, nested transactions have been used to ensure the ACID properties; i.e., atomicity, consistency, isolation and durability of updates. In HIDRA, it is required that replicated objects follow a layered structure [5] where an object can only use the services provided by objects placed below it. This ensures that no invocation cycle may arise when invocation chains are considered. As a result, no deadlock can occur due to concurrency control considerations.

So, compared to nested transactions, our ROI mechanism has the following differences and additional properties:

- Relaxed isolation. Since no deadlock may happen and an invocation is never rolled back, the updates made by a given ROI may be immediately known by other ROIs.
- Forward progress. An invocation cannot be aborted once it has modified the state of an object replica.
- Client transparency. Programmers of singleton client objects do not see any difference between an invocation to a single object and an invocation to a replicated object.
- The additional objects needed in the ROI protocol and all protocol steps have to support the failure of any ROI agent.
- Retained results. The results of a ROI have to be retained by the replicas of the invoked object until all replicas (if any) of the client object have gotten them. Thus, if the primary replica of a client fails, another replica can get the results without needing a reexecution of the method previously invoked by the faulty client replica.

The solution proposed in HIDRA for its ROI mechanism relies on some additional objects that are created each time a replicated object is invoked. These objects are needed to manage the termination of the invocation in all replicas of the invoked object:

- **RoiID**. Our support creates an object of this kind in the client domain each time a ROI is initiated. This object identifies the ROI, being needed to detect retries of the ROI in case of failures of the client or the coordinator (See Section 3 for details on ROI agents).
- **TObj**. The TObj is needed to detect when all replicas of the invoked object have been consistently updated and have terminated the invocation. It is mainly needed by our concurrency control mechanism [11].
- **CObj**. The CObj is a replicated object that is initially created in the coordinator domain and is maintained by our ORB support. The CObj is needed to detect when the retained results and the rest of the ROI context may be safely discarded.

The ROI protocol is similar to a simplified commit protocol where no rollbacks may arise. HIDRA provides other mechanisms that have been used to develop this protocol. First, it provides an *unreferenced notification*. When either a single or a replicated object has lost all its client references, it receives an asynchronous unreferenced notification. In case of a replicated object, all its replicas receive it. Also, if a client tries to *invoke a crashed object* and this object is replicated, our HIDRA support on the client side detects this situation and reinitiates the invocation on another object replica. If no other replica may be found, the client object receives an exception that notifies the failure of the object.

3. Reliable Object Invocation Agents

We call *agents* all components of a ROI that have a different role in this mechanism, and which participate in the ROI completion. In the coordinator-cohort model we use four agents:

Client. It is the domain which initiates the invocation on the replicated object.

Coordinator. It is the object replica which receives and processes locally the invocation made by the client. As the invocation proceeds, it may perform several checkpoints on the cohort replicas.

Cohort. Each one of the object replicas which has not received the client invocation and that will receive the checkpoints initiated by the coordinator.

Service Serializer (SS). It is an agent needed in this replication model to guarantee that all invocations are only allowed to proceed when no other conflicting invocation is being processed in the rest of object replicas.

In the passive model, only an object replica (the primary) receives all client invocations. In this replication model no service serializer is needed, since a local concurrency control mechanism may be used by the primary replica. The protocol described in this paper assumes the coordinator-cohort model.

4. ROI Mechanism

The basic ROI mechanism is depicted in figures 1 through 3, where the rectangles represent object references

and the boxes with rounded corners represent object instances. A ROI consists of the following steps (the agent which initiates the step is cited in parentheses):

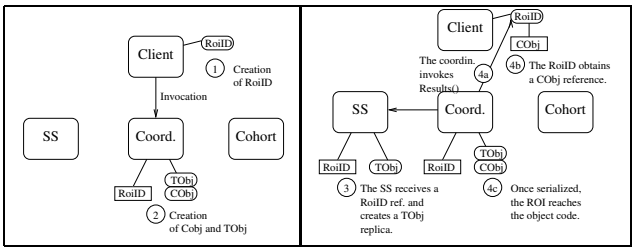

Figure 1. Steps 1 through 4 in a ROI (assuming single client).

1. *Creation of the RoiID* (Client). If the client is replicated, a special synchronous checkpoint is needed before the invocation is initiated. In this checkpoint, the RoiID is created by our ORB and it is transferred, with the RoiID assigned to the current method execution in the client, to the client replicas which will reuse it in case of failure.

   If that checkpoint was not made, the RoiID is created now and marshaled. In this case, a SINGLETON flag is set in the invocation stream to indicate that the client is not a replicated object.

2. *Reception of the invocation stream* (Coordinator). The replica chosen as the coordinator for this ROI receives the invocation stream and unmarshals the RoiID reference. The invocation is still not delivered to its target object.

   If the SINGLETON flag is not set, this coordinator checks if the current ROI is a replay of another previous one. To this end, it tries to match the received RoiID reference among the buffered ROI contexts and ROI results associated to already terminated invocations (see steps 6 and 7) whose replicated client failed. If this context is found, the results of the previous attempt are gotten and the reply is immediately returned to the client which retakes the protocol in step 8 (This is needed to discard the retained results in all server replicas).

   If the context is not found, the CObj and TObj first replicas are created in the server coordinator domain.

3. *Serialization request* (Coordinator). The coordinator makes a `Serialize()` request to the SS. The SS receives RoiID and TObj references and it creates a replica of the TObj. The internal reference of the TObj replica in the SS is immediately released. Thus, when the ROI is completed in the server side, this object receives an unreferenced notification.

   The serializer blocks the current ROI until all its predecessor RoiIDs have been reported as completed. Once this happens, the SS terminates the serialization request and this ROI is allowed to continue.

4. *The coordinator invokes the* `Results()` *method of the RoiID* (Coordinator), transferring a CObj reference if the client is not replicated (otherwise, this invocation is not needed). Later, it invokes its actual object code. Once this code has been invoked, it may initiate checkpoint invocations to the cohort replicas.

   A RoiID reference is needed in all checkpoints to identify the ROI associated to that checkpoint.

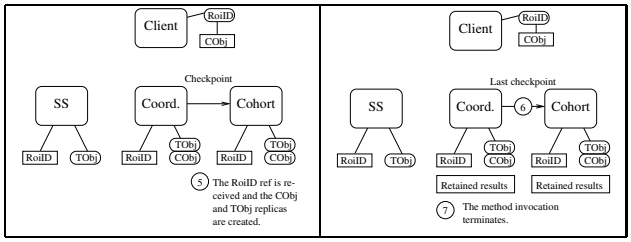

Figure 2. Steps 5 through 7 in a ROI.

5. *The first checkpoint is made* (Coordinator). A CObj and a TObj references are included in the checkpoint message by our checkpoint support. They are needed to create their replicas in the cohort domains. Moreover, a copy of the invocation arguments received by the coordinator is also sent to the cohorts.

6. *The last checkpoint is made* (Coordinator). When a cohort receives the last checkpoint for the ROI, it saves a copy of the results in the *retained results* buffer, associating them to the appropriate RoiID and CObj. It also releases the internal TObj reference of its replica. The internal CObj reference is only released if the client is a singleton object.

7. *The method invocation terminates* (Coordinator). Once the coordinator replica has terminated the method execution, it returns control to the skeleton placed in its domain. Then, the results are also saved in the *retained results* buffer and the reply is returned to the client. It also releases the internal TObj reference of its replica, which raises the unreferenced notification for this replicated object (see step 9). The internal CObj reference is only released if the client is a singleton object. Otherwise, a CObj reference is included in the return-of-invocation message.

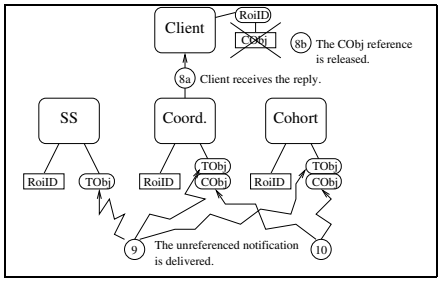

Figure 3. Steps 8 through 10 in a ROI.

8. *The client receives the invocation reply* (Client). If the client is not replicated, the RoiID releases the CObj reference. This originates an unreferenced notification to the CObj replicas.

   If the client is replicated, it has to make a synchronous checkpoint which transfers the received CObj reference and the invocation results to the client replicas. Later, the client invokes the oneway `DiscardResults()` method of the CObj. This invocation is needed because we must guarantee that the retained results are not discarded until all client replicas have gotten these results.

   All client replicas release their internal RoiID references.

9. *The TObj replicas in the serializer, coordinator and cohorts receive the unreferenced notification* (Replicas and SS). When this happens the ROI has terminated on the server side. So, the serializer tags the ROI as terminated and destroys its context. Also, the replicas destroy their copy of the input arguments and their TObj replicas.

10. When the CObj replicas receive the asynchronous unreferenced notification or the `DiscardResults()` invocation, all the context associated to this ROI (retained results, RoiID and CObj) is discarded.

    Finally, when the RoiID references have been released in all server replicas, the RoiID receives the unreferenced notification and it is destroyed.

As a consequence of this last step, the invocation has been successfully completed and all the object replicas have updated their state accordingly.

5. Failure Analysis

This section describes the behavior of the ROI protocol when one or more than one of its agents fail. Depending on the protocol step where the failure arose, different actions may be appropriate. So, each failure case is decomposed according to the step where it happened. Moreover, the failure cases are divided in *single* and *multiple* failures according to the number of agents involved in the failure.

5.1. Single Failures

We assume the SS is replicated in such a way that it only fails when the whole service has failed [11]; i.e., when at least a replica of the service objects remains alive, the SS is able to serve its requests. So, our protocol has only to deal with the failure of the client, the coordinator or any of the cohorts of a given ROI.

5.1.1. Client Failure

The management of client failures depends on the type of client (either single or replicated). So, both cases are discussed separately.

Single Client. If the client fails once the step 1 has been completed, no special action has to be taken by the rest of ROI agents. Interaction with the client is only needed in step 4 —when the `Results()` method of the RoiID is invoked— and in step 8 —the client receives the invocation results—. In step 4, the coordinator receives an exception as the result of the `Results()` invocation, since the RoiID no longer exists. This exception can be ignored, and the protocol goes on. Later, in step 8 the results cannot be delivered, but this does not matter. Finally, the release of the CObj reference in the client domain was implicitly made when the client failed. As a consequence, step 10 is reached without needing any client activity.

Replicated Client. If the client fails before the first step has been completed, then if at least the initial synchronous checkpoint has arrived to one of its replicas, that replica will be able to repeat later the invocation using the same RoiID.

Once step 1 has been completed, the failure of the client implies that a reconnection has to be made by any other client replica to collect the results produced by the first attempt. The retry will use the same RoiID, so it will be easily identified as a retry and it will get the retained results, if any, or it will wait for these results. Once the results are gotten, the protocol follows as described in step 8. Then, the new client replica has to explicitly invoke the `DiscardResults()` at the end of step 8, to discard the retained results of the server replicas. Note that the TObj unreferenced notification was raised in the first attempt and that this allows other conflicting ROIs to proceed, independently of how long the retained results are maintained.

5.1.2. Coordinator Failure

The failure of the coordinator is always detected by the client, whose support receives the notification of the failure. Then our client support has to choose another coordinator replica and invoke it again.

If the coordinator failure happened before step 3 the protocol is reinitiated from the start, reusing the same RoiID. No special action is needed, since no other agents know about this ROI.

If the failure happens after the serialization request but before the first checkpoint is made, the SS has already ordered the ROI. Since the ROI is restarted using another coordinator, a different TObj and CObj are created. As a result, if the client is not replicated, the RoiID has to ignore the `Results()` invocation made using the old CObj, releasing that CObj reference. This happens when a new invocation to the `Results()` method arrives carrying the new CObj reference. Also, the SS does not try to associate the new serialization request to the same ROI. So, the new attempt is serialized again. The previous one was tagged as terminated when the TObj received the unreferenced notification. These actions are depicted in Figure 4.

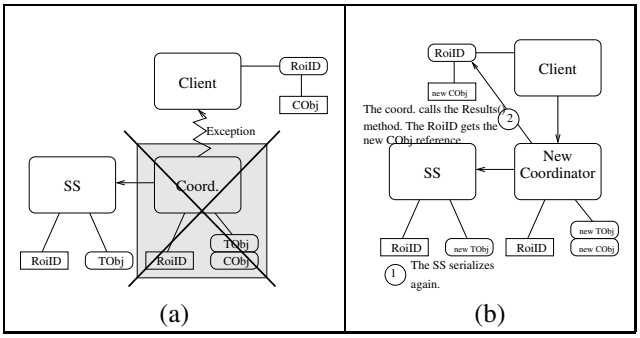

Figure 4. Failure of the coordinator in steps 3 or 4. (a) Initial situation. (b) Reattempt after failure.

Once the first checkpoint is made, if the coordinator fails, the new coordinator replica does not need to make a serialization request because it already has CObj and TObj replicas and knows that this ROI was already serialized, as it is depicted in Figure 5. In this case, when the ROI reaches the new coordinator (step 1 in the figure), the invocation continues from the point received in the latest checkpoint (steps 2 and 3a in the figure).

If the last checkpoint was also sent by the crashed coordinator, the new one has already finished the ROI and maintains the results and output arguments. Note that in the case of a singleton client we need to transfer a CObj to the client domain as soon as possible (step 4 of the protocol) to ensure that the CObj does not receive an unreferenced notification if the coordinator fails once the last checkpoint has been made and before the results are returned to the client. Thus, we ensure that the results are still retained in this situation and that the client is able to get them in this retry.

So, when the coordinator checks if this RoiID is already known (step 2), it will find that the ROI has terminated and it will return its buffered results immediately (step 3b).

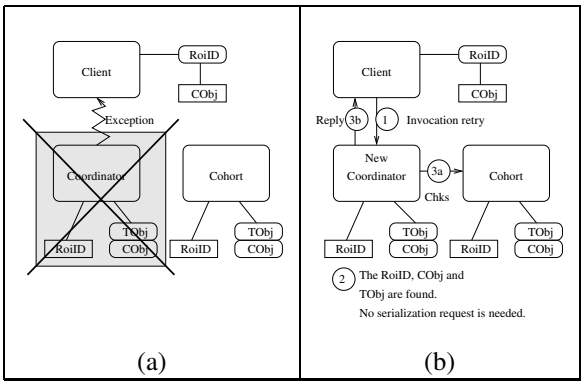

Figure 5. Failure of the coordinator after step 4. (a) Initial situation. (b) Reattempt after failure.

5.1.3. Cohort Failure

When a cohort failure arises, it is detected by our support which updates the information needed to do the checkpoints in this service. Once this action is done, the ROI is allowed to progress normally.

5.2. Multiple Failures

When the failure of multiple agents in a ROI is considered, several cases arise which may include different agents in a chain of nested ROIs. The failure of one or more cohorts is not problematic and it can be managed as it was explained in Section 5.1.3.

So, the cases which have to be dealt with are the simultaneous failure of a coordinator and its client and the failure of all the replicas which compose a service and whose failure may break a chain of nested ROIs. These two cases are outlined in the following sections.

5.2.1. Coordinator and Client Failures

When the coordinator and client agents fail before the first checkpoint has been made by the coordinator, the SS may have already ordered the current ROI (if step 3 was reached), and if this was done, the SS has lost its TObj client reference. So, when the ORB reconfigures its state and rebuilds the reference counts, the TObj replica of the SS receives an unreferenced notification, and the SS assumes that this ROI has been terminated. If the client was a replicated object, the ROI will be reinitiated on another coordinator replica. But this reattempt will use a different TObj object, and it will be serialized again (and as a different ROI) by the SS.

A different case arises when the failure happens once at least the first checkpoint was made by the crashed coordinator and before the last checkpoint was terminated (step 6 of the ROI protocol). Since the first checkpoint also carries a copy of the input arguments of the invocation, all live cohort replicas are able to continue the ROI, but one of them has to be chosen as the new coordinator.

In this case, the TObj replicas do not receive any unreferenced notification since its replicas in the cohort domains have not released their client reference. So, the remaining cohorts have to cooperate to successfully terminate the ROI. To this end, a new coordinator replica is chosen in the HIDRA reconfiguration steps and it will resume the ROI. As a result, the resumed ROI eventually reaches step 7. Since the client failed, the results cannot be delivered to it. But a copy of them is retained in all replicas of the invoked object. This copy is released if the client was not a replicated object when the unreferenced notification arrives to the CObjs. Otherwise, it is maintained, waiting for a replay of the ROI made by another client replica.

Finally, if this multiple failure arises after step 7 and the client was not replicated, all cohorts finalize and release the ROI context and buffered results; otherwise, the results and the ROI context are maintained until another attempt is reissued by one of the client replicas.

5.2.2. Service Failures

If an entire service fails, probably more than one chain of nested ROIs will be broken. As shown in Fig. 6, the client of the failed server will receive an exception as a result of the invocation.

If the failed object was also the client of a replicated object, then this invoked object will maintain the results of that ROI indefinitely. To avoid this situation, once the TObj has been discarded, our support has to check periodically if the RoiID references maintained in the server replicas are still valid. To this end, they can use the standard `CORBA::object::non_existent()` operation which returns TRUE when the client reference points to an object which no longer exists. Once these replicas have noticed that all client replicas have crashed, the retained results can be discarded.

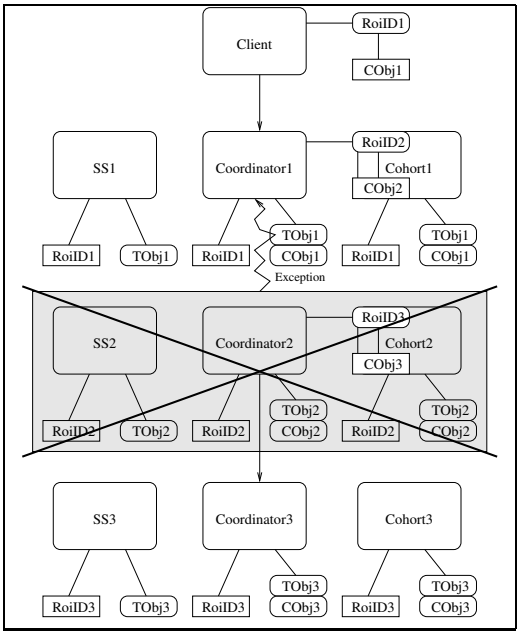

Figure 6. Failure of all object replicas.

6. Related Work

Replication of software components is a common way to achieve high availability. Multiple techniques have been used to ensure the consistency of the replicas, and they are based either on atomic multicasting, transactional support or some kind of checkpointing. The solution varies according to the replication model being used.

Our ROI mechanism is conceived to ensure the consistency when a replicated object is invoked. Our system design guarantees that an initiated invocation always is completed by all the object replicas and that when a failure arises and a reply is needed, the invocation is able to collect the results of the previous attempt or to continue its actions from the failure point. Moreover, the coordinator-cohort and passive models being used permit some load balancing if we vary the coordinator or primary replica chosen at each invocation.

The systems based on an active replication model [12] need to use some atomic multicast protocol [1, 3, 13] to ensure that all replicas receive the same requests in the same order. However, this replication model needs that each replica processes locally all requests, updating accordingly their state. Although each replica does not need to know the existence of the rest, this approach cannot be used to balance the load of a distributed system. Moreover, the protocols needed for atomic multicast need some delays and messages to ensure that the appropriate order is followed. There have been a lot of systems which follow this model [4, 9]. Some of them [4] also need some kind of transactions to add more replicas to the replicated object, incurring in additional costs. We consider this model too expensive —in terms of messages and computing power— and inflexible —because the management of a shared resource by all object replicas seems complicated.

The original design of the coordinator-cohort model [2] also associated an identifier to each incoming request and maintained the *retained results* to avoid reexecutions of the request in case of failure. But its solution does not give a

practical rule about how (and when) retained results can be discarded. Its solution only relies on the synchrony of the needed checkpoints. Moreover, it does not consider the failure of the whole replicated object and its consequences on the objects it previously invoked, which maintain these retained results indefinitely. The only solution provided for this case consists in making the checkpoints in stable storage, too. Later, some replicas have to be recovered and they have to reinitiate the invocation once they have read their state.

Finally, another technique to ensure the consistency of the object invocations is the use of nested transactions and some concurrency control mechanism as in [6, 7, 8]. The support for transactions should be included in the object invocation mechanism to be transparent for the programmer, and this does not happen in [6] nor in [8]. Also, some care must be taken to avoid deadlocks or to detect them and abort the appropriate transactions.

7. Conclusions

The *reliable object invocation* mechanism described here provides the basis to ensure the consistency of the replicated objects supported by our HIDRA architecture. The context associated to each invocation is created and propagated by some components of the HIDRA ORB, so this mechanism is transparent for the programmer of client objects.

The objects involved in a ROI allow that each agent knows immediately when the ROI has been successfully terminated and they also make possible a fast detection of the failure of any agent, ensuring that the protocol manages correctly that situation.

This invocation mechanism can be used in the passive and the coordinator-cohort replication models, it requires few additional messages to ensure the consistency of all replicas and it also supports seamlessly the failure of several of its agents.

Finally, the ROI mechanism also provides a cheaper invocation support than those used in active replication models, which need atomic multicast protocols. Additionally, the replication model assumed in the ROI can be the basis to balance the load of the distributed system, choosing at each time the more appropriate coordinator replica.

References

[1] Ö. Babaoğlu and A. Schiper. On group communication in large-scale distributed systems. In *Proc. ACM SIGOPS European Workshop, Dagstuhl, Germany*, volume 29(1) of *ACM Operating Systems Review*, pages 62–67, 1995.

[2] K. P. Birman, T. Joseph, T. Raeuchle, and A. El Abbadi. Implementing fault-tolerant distributed objects. *IEEE Trans. on SW Eng.*, 11(6):502–508, June 1985.

[3] K. P. Birman and R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, Los Alamitos, CA, 1994.

[4] E. C. Cooper. *Replicated Distributed Programs*. PhD thesis, Univ. of California, Berkeley, CA, April 1985.

[5] P. Galdámez, F. D. Muñoz-Escoí, and J. M. Bernabéu-Aubán. High availability support in CORBA environments. In F. Plášil and K. G. Jeffery, editors, *24th Seminar on Current Trends in Theory and Practice of Informatics, Milovy, Czech Republic*, volume 1338 of *LNCS*, pages 407–414. Springer Verlag, November 1997.

[6] S. Ghemawat. Automatic replication for highly available services. Technical report, MIT-LCS-TR-473, MIT Lab. of Comp. Sc., January 1990.

[7] T. Hirotsu and M. Tokoro. Object-oriented transaction support for distributed persistent objects. In *Proc. of the 2nd International Workshop on Object-Orientation in Operating Systems*, September 1992.

[8] M. C. Little and S. K. Shrivastava. Replicated K-resilient objects in Arjuna. In *Proc. of IEEE Workshop on the Management of Replicated Data, Houston, Texas*, pages 53–58, November 1990.

[9] S. Maffeis. *Run-Time Support for Object-Oriented Distributed Programming*. PhD thesis, Dept. of Comp. Sc., Univ. of Zurich, February 1995.

[10] J. E. B. Moss. Nested transactions: An approach to reliable distributed computing. Technical report, MIT/LCS/TR-260, MIT Lab. for Comp. Sc., 1981.

[11] F. D. Muñoz-Escoí, P. Galdámez, and J. M. Bernabéu-Aubán. HCC: A concurrency control mechanism for replicated objects. In *Proc. of the VI Jornadas de Concurrencia, Pamplona, Spain*, pages 189–204, July 1998.

[12] F.B. Schneider. Replication management using the state-machine approach. In S. J. Mullender, editor, *Distributed Systems (2nd ed.)*, pages 166–197. Addison-Wesley, Wokingham, UK, 1993.

[13] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible group communication system. *Communication of the ACM*, 39(4):76–83, April 1996.