Extending an ORB to Support High Availability

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

An architecture to support the development of highly-available applications is proposed. An ORB is used as a base to build object-oriented applications and to solve their intercommunication needs. The high availability architecture extends the basic ORB services, adding support for object replication using multiple primary and secondary replicas. Each primary can serve directly any arriving request, checkpointing the resulting state to the rest of replicas. Also, this architecture offers mechanisms to guarantee the consistency of the state of a replicated object. These mechanisms are the use of transactions to manage the arriving requests to the replicated object and the use of a synchronization component to control which requests may proceed.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*network operating systems*; D.4.5 [**Operating Systems**]: Reliability—*fault tolerance*; D.4.7 [**Operating Systems**]: Organization and Design—*distributed systems*

General Terms: Distributed systems, Reliability

Additional Key Words and Phrases: Distributed algorithms, group communication, high availability, fault tolerance

1. INTRODUCTION

Distributed systems are a good basis to support highly available applications. In a distributed system there are multiple nodes which may have independent behavior when failures arise. So, if some support is given by the underlying operating system, a programmer can build applications with independent and replicated components placing them in different nodes of the system. This kind of applications may be considered highly available.

Replication support is not the only requirement to develop highly available applications in a distributed system. The programmer has to decompose its application in different software modules which require communication primitives; i.e., he has to develop a *distributed application*. If the programmer plans to build an object-oriented distributed application, CORBA [OMG 1995a; OMG 1995b] provides him the support needed to interconnect the application components, using object invocation services. Our aim is to extend an ORB to provide also support for highly available object-oriented applications.

Besides deciding the communication support used to interconnect the application components, an object replication scheme has to be chosen. *Active replication* [Schneider 1993] assumes that each request set to the replicated service is received and processed by every replica. The output results produced by each replica are collected, compared and somehow merged before they are returned to the client which made the request. *Passive replication* [Budhiraja et al. 1993] relies on an

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

active or primary replica which receives all the requests and directly updates its local state. This active replica checkpoints the state updates to a set of secondary or backup replicas which are copies of the primary one and that will take its role in case of failure.

Usually, when the support for high availability has been provided by the distributed operating system, passive replication has been the chosen alternative [Allchin 1983; Borg et al. 1989]. This kind of replication has the advantage that only one node of the system has to serve the request. On the other hand, group communication toolkits [Babaoğlu et al. 1994; Birman and van Renesse 1994; Malloth et al. 1995; van Renesse et al. 1995] exist that can be used on top of an operating system to develop highly available applications. These toolkits usually offer active replication, since group communication primitives are used to order and multicast requests to all replicas. Other approaches, as the mechanisms presented in [Ladin et al. 1992] or [Birman et al. 1985] allow a type of replication that can be considered intermediate, either because the number of replicas that can behave as primaries is configurable or because the primary replica varies for each request.

Our solution tries to provide high availability support as a service integrated at kernel level, extending an ORB. These extensions to the ORB focus on providing support for object replication based on checkpointing and the use of transactions to improve the reliability of the requests made to the highly available objects. Our replication scheme uses a primary-backup approach, but allowing multiple primaries for the same replicated object. Each request can be directed to a different primary replica, which once it has gotten control of the object may do checkpoints to the rest of the replicas until the request finishes. This is similar to the *coordinator-cohort* approach described in [Birman et al. 1985].

A concurrency control mechanism based on a compatibility matrix which determines what requests to a replicated service can be executed concurrently is also used. Thus, when multiple primary replicas exist for a service, several compatible requests may be concurrently served in different nodes and the consistency of the replicated object state is still guaranteed. This is a good property of our support that is not present in other architectures, as the one shown in [Beedubail and Pooch 1996]. Other approaches [Bouwmeester et al. 1993] rely on distributed locks, which need higher communication costs.

The rest of the paper is structured as follows. Section 2 explains the model of highly available objects provided by this extended ORB, whose original architecture is given in Section 3. Section 4 outlines which are the additional components required to provide these new services. Later, section 5 describes the ORB extensions to manage replicated objects and services and to support checkpointing and transactions. Section 6 presents how some kind of failures are supported by this extended architecture. Finally, section 7 summarizes the main characteristics of this support for high availability and the work that still has to be done in this area.

2. HIGH AVAILABILITY MODEL

Our support for high availability manages the concept of *replicated services*. A service, which is composed by a set of objects is entirely replicated on different domains. Each domain may be either a *primary replica* or a *secondary replica* of the service. For each object in the service, primary domains hold primary object replicas while secondary domains hold secondary replicas.

Primary replicas provide publicly available interfaces and are able to serve client invocations to these interfaces. On the other hand, secondary replicas are not directly accessible by clients and only maintain a copy of the state of the primary replicas.

Multiple replicas of each kind are allowed for the same object. If several primary replicas exist, all of them may receive client requests, but all these requests can not be served at the same time, since they might introduce inconsistencies among the states maintained in different replicas. So, for each request, the primary replica that receives the invocation becomes the *coordinator*. It serves the request and updates its local state accordingly. Before providing a response to the client, the coordinator has to do at least a checkpoint to the rest of object replicas, which are the *cohorts* for this request. But some concurrency control mechanism is needed to guarantee that different coordinators do not modify concurrently the same part of the object state in different replicas, because otherwise the shared state could not be maintained consistent when their checkpoints were received by the rest of replicas. To provide this concurrency control management a special *serializer* object exists in each replicated service. This serializer object has a compatibility matrix where information about which service operations are compatible is maintained[1]. Before a coordinator begins to execute the operation that it has to serve, it has to request to the serializer if it can proceed. As a result, the serializer returns the identifiers of the transactions that must precede the current one. When all these transactions have been terminated on the node where this request was made, its operation is allowed to proceed.

To guarantee the reliability of the invocations made to a replicated object, our architecture offers transactions. Each time an invocation is initiated, a transaction is created and it will be committed when the invocation terminates and all object replicas have updated its state. If some node crashes when the transaction proceeds, the transaction information maintained by the rest of the nodes allows its completion when the system is reconfigured.

3. OBJECT REQUEST BROKER

The ORB used as the basis of our support for high availability has a design similar to the one described in [Bernabéu-Aubán et al. 1996]. This design divides the ORB in several layers, which are traversed in descending order from the client side of an object invocation and in ascending order in the server side, as shown in figure 1.

The bottom layer provides a *reliable transport* service, needed to transfer messages among the ORB components placed at different nodes. Its reliability is guaranteed by a *cluster membership monitor* (CMM) [Muñoz-Escoí et al. 1997] which is a component that checks and maintains the current set of nodes that constitute the system. Thus, when a message is sent to a remote node, the transport layer ensures that it is delivered to its target or an error is returned to notify that the target node has crashed. The CMM, in case of changes in the membership set, besides notifying the ORB transport also coordinates all ORB components in some steps needed to

[1]Two operations are considered incompatible when at least one of them modifies a part of the object's state that is read or modified by the other one.

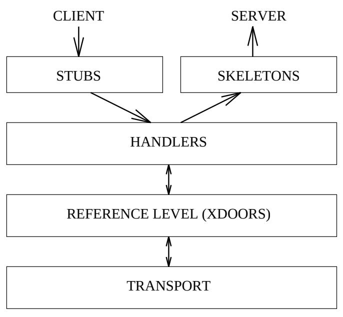

Fig. 1. Organization of the ORB.

reconfigure the system.

The other layers model object invocations. They use the services provided by the transport to guarantee that all initiated invocations either are successfully completed or raise an exception indicating that the node where the target object resided has crashed. These layers are the *xdoor* layer, which manages the references needed to invoke objects and the *handler* layer, that are used to deal with marshal streams, marshaling the outgoing arguments, and unmarshaling the incoming ones.

Xdoors have different roles, they can be either client xdoors or server xdoors, depending on the domain that uses them. There is one server xdoor associated to each object, and it constitutes the target entity for all incoming invocations to the object it represents. A client xdoor is a reference to the server xdoor that is used to make invocations to its object from other nodes. There is only a client xdoor for each node with client domains for the object being referenced.

Handlers are placed on top of xdoors and are mainly used to marshal object references when the stream of information to be sent or returned is being built. Conversely, they also have to unmarshal the data contained in incoming streams. Besides this, handlers maintain a count of the references for its object that exist in its local domain. Two types of handlers exist, one for the client domains and another for the server domain that maintains the object implementation. In a given node there are as many handlers as domains allowed to access the object. Thus several handlers may share the same xdoor.

Both, handlers and xdoors participate in the protocol needed to maintain reference counts. The reference count is used to know when there are no references for the server object. In that case, the implementation of the object receives an unreferenced notification, and it can release its resources and terminate itself.

To extend the functionality of the xdoor and handler layers, new subclasses of these two entities have to be implemented, providing the extensions needed.

In case of failures, the ORB has to guarantee that its state is reconfigured appropriately. This means that all live nodes have to know which are the nodes that have crashed and the reference counts have to be recomputed. To do so, the CMM provides the identifier of the crashed nodes. Thus, all invocations made to objects placed on them are aborted or return an exception. To rebuild the reference counts, a protocol is run among the live nodes with client xdoors for an object and the node that maintains its server xdoor.

4. ADDITIONAL COMPONENTS

The extensions made to the ORB presented in the previous section require some additional components that deal with some characteristics of replicated objects and services that can not be found in single objects. These components are the service manager and its agents.

A *service manager* (SM) is needed for each one of the replicated services. It knows where are placed the service replicas and which is the class —either primary or secondary— of each one of them. Also it provides concurrency control services to guarantee an appropriate execution flow of the incoming service requests; i.e., it acts as a *serializer*. To this end, it maintains a copy of the compatibility matrix.

There is only one instance of each SM in the whole system, which is placed in a node where at least exists a primary replica of its service. Other replicas of this SM exist, but they only maintain part of its data —mainly, the compatibility matrix. The rest of its state can be rebuilt in case of failure from the state of its remaining service manager agents.

Service manager agents (SMA's) are placed in all the nodes where some replica or client of the service exists. The SMA performs all the service management operations that may be locally completed, mainly it guarantees that the execution order chosen by the SM is respected by the requests that arrive to the local node. Also, it has to know where are placed all other replicas of this service.

5. ORB EXTENSIONS

The current section outlines the extensions made in the ORB to provide high availability services to the rest of the system. A deeper explanation can be found in [Galdámez et al. 1997] where the whole architecture is described. Here, only the extensions made to create, identify and manage replicated objects are described. Also some details are given about how checkpointing invocations are made and how transactions are managed.

5.1 Replicated Objects Management

Compared to the original ORB described in section 3, our proposal has other components to manage an object invocation. The first important addition is the *replicated object xdoor* (ROX), which is a new type of xdoor that represents a replicated object. There is, at most, a single ROX for a given replicated object in each node. Since an object may have different replicas in the same node or different client domains with access to the object, the ROX may be attached to multiple handlers. Handlers for highly available objects are a bit different to standard handlers. So, a new type of handlers is needed, it is the *replicated object handler* (ROH) class. The ROX labels each of the ROH's which it is attached to, either as primary, secondary or client.

The ROX's are also highly related to the SMA's located in their nodes. These SMA's are used to register the creation of the highly available object replicas and to manage the first steps of all incoming invocations to these objects. ROX's maintain as its local state the identifier of the object they represent. This identifier is composed by a pair (ServiceId, ObjectId) whose numbers are generated locally by each SMA but are global and unique in the whole system. Since ROX's constitute a new type of xdoor, the ORB may take some different procedures to manage them. This approach has been used to provide a different reference counting protocol for them.

In fact, when the first instance of a replicated object is created, the first ROX for it is created in the local node. The ORB is requested to register the object creation and its object identifier is generated and attached to this first ROX. Also, the host domain obtains both a server and a client ROH which will be used to process the local and incoming invocations. If a replicated object reference has to be passed to another node or domain, in the marshaling process a representation of the object is included. This representation consists of the xdoor type identifier for ROX's and the system-wide object identifier. When this information is received in the target node, it is unmarshaled. To do so, once the system-wide object identifier is gotten, the ORB checks if it already exists in its node. If not, a ROX is created and a newly created client ROH is attached to it. Otherwise, the received ROX replaces the old one. In any case, if the receiving domain is a replica for the same service and it has to create a replica for the incoming object, the SMA is requested to register the new replica and a server ROH is created and attached as an upper handler for this ROX.

To deal with unreferenced notifications in replicated objects, it is needed that **all object replicas** receive this notification when no client references exist in the system. To this end, the oldest ROX, known as *main ROX*, for the replicated object maintains the reference count.

In case of failure of the main ROX, in the reconfiguration phase a new main ROX is chosen by the SMA's. Later, in the final steps of the reconfiguration phase its reference count is updated according to the information sent by the rest of the nodes with client ROX's.

Finally, another important aspect related to replicated object management is how these objects are invoked. To be able to invoke a replicated object, the client domain must have a client ROH which points to a client ROX. The invocation is managed by these two components and the client ROX uses the services provided by the local SMA to choose a target node with a primary replica. The message with the invocation information is transferred to that node, where its ORB finds its target ROX and transfers the incoming message to it. So, the message routing is done by the SMA's of the service being used.

5.2 Checkpointing

Each service replica has an special object of the checkpoint class. This checkpoint object, compared to other objects in the service, has different characteristics. First, it must be only invoked from other objects in the same service. Second, it is created as soon as the service is created and installed. Third, all the invocations being made to a checkpoint object have to be multicast to all its replicas. Finally, it does not receive unreferenced notifications since it will be destroyed when the service ceases to exist.

Since it has a different behavior, the checkpoint object requires different handlers and xdoors, compared to the rest of replicated objects described in the last section. So, it uses a Chkhandler at the handler layer and a Chkxdoor at the xdoor layer.

When the checkpoint object is created, its host domain receives both a client Chkhandler and a server Chkhandler, attached to the Chkxdoor.

When a checkpoint has to be made, the primary replica invokes one of the checkpoint methods. When the invocation stream reaches the Chkxdoor, using the services provided by the local SMA, transfers a copy of the stream to each remote node with replicas of the checkpoint object, and to other local domains with replicas, if any. The Chkxdoor updates accordingly the reference counts of all object references being transmitted.

When the ORB's placed in the destination nodes receive the invocation message, they check the target object identifier to locate the Chkxdoor that has to be used to transfer the invocation to the associated server Chkhandlers. However, the stream is inspected to find out if this is the last checkpoint in a transaction or not. If it is not the last one, the invocation is buffered and is not delivered to its target object until the last one arrives. This permits to rollback easily a transaction.

In case of failure of the coordinator replica for a transaction when some checkpoints are already buffered but the last one still has not been sent, the transaction is reinitiated. The new attempt chooses a new coordinator replica and its new checkpoints will overwrite all the previously buffered ones.

5.3 Transactions

Each invocation made to a highly available object is done in the context of a different transaction. Transactions are used to guarantee that all attempted invocations are completed in all of the object replicas or do not modify the state of any of them. Furthermore, transactions are the units used to manage the concurrency control mechanism, which is based on a *serializer* object for each service, that decides the transactions that can proceed using an incompatibility matrix. This incompatibility matrix maintains information about which publicly accessible operations of the objects that compose the service can not be executed concurrently.

Transactions are initiated by the SMA of the client node which tries to invoke one of the operations of a replicated object. This SMA creates a TID object and includes a client reference to it in the information to be transferred to the target object. This TID identifies the new transaction and is used to detect the failure of the client node.

Once the invocation arrives to the SMA of the node where the target object is placed, this coordinator SMA creates a *confirmation object* (CObj) that is used later to notify the end of the transaction. Before the transaction arrives to the code it wants to invoke, the coordinator SMA collects the CObj reference and enough information to identify the operation being invoked and calls the serializer object. The serializer creates a replica for the CObj that later uses to know when the current transaction has ended. Moreover, the serializer checks the incompatibility matrix to find out which are the operations incompatible with the current one and consults the list of still not finished transactions to return to the requesting SMA which are the transactions that have to terminate before the current one has to be allowed to proceed.

The coordinator SMA blocks the incoming transaction until it knows that all transactions that must precede it have terminated. If the information returned by the serializer has an empty list of preceding transactions, the current one is allowed to proceed immediately.

Each transaction has to do at least a checkpoint to the rest of object replicas to report them the changes made in the object state or to notify the end of a read-only request. When the first checkpoint of a transaction is initiated by its coordinator replica, the SMA adds the TID and CObj references and a copy of the invocation arguments to the information being transferred. In this way, the other replicas know about the current transaction and about the changes it has made to the object's state. These cohort SMA's create a CObj replica as soon as they get its reference. The last checkpoint for a transaction includes a flag pointing out that it is the last one. When the cohort SMA's receive this last checkpoint, all buffered checkpoints are delivered and the object replicas update its local state. Later, these SMA's remove the current transaction from the list of precedent transactions for all the blocked ones. Once this has been done, the SMA invokes a method of the TID passing a reference to the CObj. The TID maintained by the client SMA waits for termination of the invocation made to the coordinator node. To signal the termination of the transaction to all replicas of the invoked object, the client SMA also waits for as many invocations to its TID as nodes with cohort replicas exist for this object. Once these invocations are received, it releases its CObj reference. As a consequence, all the CObj replicas eventually receive an unreferenced notification. This notification means that the transaction has been successfully ended and all transaction context can be forgotten.

Except the invocation made from the client node to the coordinator replica and from the coordinator SMA to the serializer object, all other invocations are asynchronous, using oneway methods. The same can be said about the unreferenced notifications used to point out the transaction termination. Thus, the concurrency of the steps followed to complete a transaction has been improved.

Finally, note that the basic components used in a transaction are the TID and the CObj objects. The TID is used to identify the transaction, allowing its restart in case of failures of the coordinator node, since the rest of replicas will know about it if some checkpoint has been made in its context. The CObj is used to point out when the transaction has ended.

6. FAILURE RECOVERY

To describe how these ORB extensions deal with failures, the analysis has been divided among the active components which participate in a transaction, studying how the transaction proceeds when each one of these components fails. It is also assumed that there is a failure detector, the CMM, which reports node failures to our support and that only fail-stop failures [Schlichting and Schneider 1983] may arise. The cases to be analyzed are the following ones:

—*Client failures.* If a client fails once it has initiated a transaction and no other failure arises, this transaction is completed by its coordinator node. No special action has to be taken. When the transaction is completed no answer is reported to the client. The transaction context is released, since the CObj replicas immediately receive the unreferenced notification.

—*Cohort failures.* In case of a cohort failure, only the state maintained by the SMA's for its service has to be updated to adjust the number of replicas.

—*Coordinator failures.* If the coordinator failure arises before the serializer is contacted, no special action is required. The client domain receives an exception and restarts the request using another primary replica.

If the coordinator fails once it has contacted the serializer but before it has made its first checkpoint, the client SMA will receive an exception notifying this fact in the reconfiguration phase. So, the client SMA repeats the invocation using another primary replica as the coordinator. The same is done if some checkpoint was done, but in the new transaction incarnation, all previously buffered checkpoints are overwritten by the new ones. No checkpoint delivery is done until the last checkpoint arrives, so the cohort replica objects still do not have processed any checkpoint.

In case the coordinator fails when all checkpoints are completed but before it returns the results to the client, the client reinitiates the transaction choosing a new coordinator which immediately returns the requested results, if any. As a result, the client is able to release the CObj, committing the transaction.

More failure cases and additional details about the cases shown above are given in [Galdámez et al. 1997].

7. CONCLUSIONS

The proposed architecture for high availability support in a CORBA environment allows either a coordinator-cohort approach, using multiple primaries or a primary-backup scheme if only a primary replica and several secondaries are used. Some extensions to the ORB are needed to manage references for replicated objects and to deal with the checkpoint invocations used to maintain a consistent state in all replicas. This has been accomplished providing new types of xdoors (ROX's and Chkxdoors) and new types of handlers (ROH's and Chkhandlers). Moreover, to ensure that the updates made to replicated objects do not introduce inconsistencies in the objects' state, a concurrency control mechanism and transaction support have been added.

The concurrency control mechanism is based on a serializer object per service, which orders the incoming requests according to the incompatibility of their operations. The state of this serializer is also partially maintained by the SMA's and can be rebuilt in case of its failure. The programmer of a highly available service only has to provide the incompatibility matrix, our support deals with the management of concurrency. So, the programmer has not to worry about concurrency control. The resulting mechanism only requires an additional invocation to the serializer, which is cheaper than the use of distributed locks as it has been done in [Bouwmeester et al. 1993]. Also, it allows concurrency among multiple incoming requests that do not modify the same parts of the object state, which provide better performance that the single-threaded primary-backup approach followed by [Beedubail and Pooch 1996] or all approaches based on active replication [Kleinoeder and Golm 1996; Maffeis 1995].

Transactions guarantee that all requests to replicated objects modify all the replicas, leaving them in a consistent state, or these requests are completely aborted and do not update the object's state. Our model of light-weight transactions satisfies this atomicity requirement, needing a few additional messages —only two per replica, both of them asynchronous— to notify the end of the transaction.

As a result, we have an extended ORB that supports highly available services, needing only a little quantity of additional messages to guarantee the consistency of all replicas' state, and permitting the concurrent execution of multiple requests to the replicated object in some cases. The programmer of a highly available service only has to dedicate additional efforts to build the compatibility matrix and to decide where checkpoints are needed.

Another attempt to provide high availability support in CORBA environments was made in Electra [Maffeis 1995]. His solution is based on group communication toolkits which are extended with an upper layer that provides the ORB services. This approach depends on the services provided by the group toolkit being used, which usually has protocols to guarantee an ordered delivery of messages into the group. The efficiency and failure tolerance of the ORB depends on the protocols used by this group communication toolkit. Moreover, the use of group toolkits as the base for replication implies that all messages sent to the replicated object are received by all its replicas. This may be convenient if active replication is used, but wastes an unnecessary number of messages if a primary-backup approach is followed. Other minor drawbacks of his approach reside in not providing means to detect when an object is no longer referenced by client domains and not supporting the marshal and unmarshal procedures for complex objects, so marshaling and unmarshaling tasks are left to the programmer.

Our solution has to be refined to deal with invocations among highly available objects, using support for nested transactions [Moss 1981]. Also, some improvements can be made when only a primary replica exists and all others are secondaries. In that case a lighter concurrency control mechanism will suffice.

REFERENCES

Allchin, J. E. 1983. An architecture for reliable decentralized systems. Technical report (Sept.), TR-GIT-ICS-83/23, Georgia Institute of Technology, Atlanta.

Babaoğlu, Ö., Davoli, R., Giachini, L. A., and Baker, M. 1994. RELACS: A communications infrastructure for constructing reliable applications in large-scale distributed systems. Technical report (June), UBLCS-94-15, Dept. of Computer Science, University of Bologna, Bologna, Italy.

Beedubail, G. and Pooch, U. 1996. An architecture for object replication in distributed systems. Technical report (March), TR 96-006, Dept. of Computer Science, Texas A&M Univ, TX.

Bernabéu-Aubán, J., Matena, V., and Khalidi, Y. 1996. Extending a traditional OS using object-oriented techniques. In U. Association Ed., *2nd Conference on Object-Oriented Technologies & Systems (COOTS), June 17–21, 1996. Toronto, Canada* (Berkeley, CA, USA, June 1996), pp. 53–63. USENIX.

Birman, K. P., Joseph, T., Raeuchle, T., and El-Abbadi, A. 1985. Implementing fault-tolerant distributed objects. *IEEE Transactions on Software Engineering 11*, 6 (June), 502–508.

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

Borg, A., Blau, W., Graetsch, W., Herrmann, F., and Oberle, W. 1989. Fault tolerance under UNIX. *ACM Transactions on Computer Systems 7*, 1 (Feb.), 1–24.

Bouwmeester, L. H. A., Kluit, P. G., and Ververs, F. 1993. Using atomic actions in replica groups to simplify the replication implementation. Technical report (Dec.), TR 93-78, Faculty of Technical Mathematics and Informatics, Delft University of Technology, The Netherlands.

Budhiraja, N., Marzullo, K., Schneider, F. B., and Toueg, S. 1993. The primary-backup approach. In S. J. Mullender Ed., *Distributed Systems (2nd edition)*, pp. 199–216. Addison-Wesley, Wokingham, England.

Galdámez, P., Muñoz-Escoí, F. D., and Bernabéu-Aubán, J. M. 1997. An architecture for high availability in a CORBA environment. Technical report (Apr.), DSIC-II/14/97, Univ. Politècnica de València, Spain.

Kleinoeder, J. and Golm, M. 1996. Transparent and adaptable object replication using a reflective java. Technical report (Sept.), TR-I4-96-07, Dept. of Computer Science, Erlangen-Nürnberg Univ., Erlangen, Germany.

Ladin, R., Liskov, B., Shrira, L., and Ghemawat, S. 1992. Providing high availability using lazy replication. *ACM Transactions on Computer Systems 10*, 4 (Nov.), 360–391.

Maffeis, S. 1995. *Run-Time Support for Object-Oriented Distributed Programming.* Ph. D. thesis, Dept. of Computer Science, University of Zurich.

Malloth, C., Felber, P., Schiper, A., and Wilhelm, U. 1995. Phoenix: A toolkit for building fault-tolerant distributed applications in large scale. Technical report (July), Dépt. d'Informatique, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland.

Moss, J. E. 1981. Nested transactions: An approach to reliable distributed computing. Technical report, MIT/LCS/TR-260, MIT Laboratory for Computer Science.

Muñoz-Escoí, F. D., Galdámez, P., and Bernabéu-Aubán, J. M. 1997. The group membership problem and its solutions. Technical report (Apr.), DSIC-II/8/97, Univ. Politècnica de València, Spain.

OMG. 1995a. *The Common Object Request Broker: Architecture and Specification.* Object Management Group. Revision 2.0.

OMG. 1995b. *CORBAservices: Common Object Services Specification.* Object Management Group. Revised Edition.

Schlichting, R. D. and Schneider, F. B. 1983. Fail-stop processors: An approach to designing fault-tolerant systems. *ACM Trans. on Computer Sys. 1*, 3 (Aug.).

Schneider, F. B. 1993. Replication management using the state-machine approach. In S. J. Mullender Ed., *Distributed Systems (2nd edition)*, pp. 166–197. Addison-Wesley, Wokingham, England.

van Renesse, R., Birman, K. P., Glade, B., Guo, K., Hayden, M., Hickey, T. M., Malki, D., Vaysburd, A., and Vogels, W. 1995. Horus: A flexible group communications system. Technical report (March), TR95-1500, Dept. of Computer Science, Cornell University, Ithaca, NY.