Using Semantic Knowledge of Distributed Objects to Increase Reliability and Availability

Pascal Felber, Ben Jai, Rajeev Rastogi, Mark Smith
Bell Laboratories – Lucent Technologies
600 Mountain Avenue
Murray Hill, NJ 07974, USA

Abstract

As systems become more distributed, they also become more complex. To ensure consistent execution while maximizing availability, distributed applications use various mechanisms such as replication, load balancing, and data caching. The protocols used for consistency management and component availability are traditionally instantiated by the application. However, in distributed object-based environments like CORBA or Java RMI, the infrastructure can often determine adequate protocols to guarantee liveness and safety based on the request and semantic knowledge of the application. This paper discusses how semantic knowledge of distributed objects can help implement intelligent behavior in middleware and choose optimal protocols for distributed component interactions.

1. Introduction

Although the idea of using semantic knowledge to improve performance has been investigated in the context of databases (e.g., [7, 4]), little attention has been paid to its application in the context of distributed systems. The OGS [6] distributed infrastructure introduced mechanisms for describing the semantics associated with replicated server, and let the infrastructure (rather than the application) use that information to choose the most efficient algorithm for distributed interactions. For instance, a read operation may not need to be delivered atomically by all servers, thus reducing the latency experienced by the client and the load on the servers.

In [15], Pedone and Schiper introduced a generalized version of atomic broadcast, called generic broadcast, that implements different ordering guarantees depending on the messages that are broadcast. Informally, given a conflict relation, generic broadcast behaves either as a reliable broadcast if messages do not conflict, or as an atomic broadcast otherwise.

In [11], Aguilera and al. propose a set of “thrifty” generic broadcast algorithms that are optimal in terms of resilience. These algorithms also depend on a conflict relation (symmetric and non-reflexive) to decide whether to behave like a reliable or an atomic broadcast.

In this paper, we discuss how distributed objects can be classified and described in terms of semantic properties. Such application-specific information can be used by a middleware infrastructure to transparently implement advanced functionalities like load balancing, replication, or caching, and to infer the conflict relations used by generic broadcast algorithms to optimize client/server interactions.

The rest of this paper is organized as follows. Section 2 presents the motivation of this paper and introduces the notion of generic algorithms. Section 3 illustrates through an example the impact of semantic knowledge in a distributed context. Section 4 specifies the different semantic properties that we have identified as useful. Section 5 discusses our vision of intelligent middleware and how it can take advantage of these semantic properties to increase reliability and availability of Internet applications. Finally, Section 6 concludes this paper.

2. The Need for Generic Algorithms

2.1. On Ordering and Reliability

Distributed computing is one of the major trends in the computer industry. As systems become more distributed, they also become more complex and have to deal with new kinds of problems, such as partial failures and link failures. To facilitate the development of distributed applications, communication primitives generally provide qualities of service defined in terms of reliability and ordering. Informally, reliability ensures that messages are not lost, and ordering guarantees that messages are not re-ordered during transmission.

Messages can be sent to a single process (point-to-point communication), or to multiple processes at once ("one-to-many" communication). There are two types of one-to-many communication primitives: broadcasts that target all processes in a system, and multicasts that target subsets of processes. Such primitives greatly simplify the development of several types of applications that have requirements for high availability, fault tolerance, parallel processing, or collaborative work.

Reliability and ordering have a high importance in one-to-many communication primitives, because they set bounds on how message delivery can diverge from one process to another. Reliability defines the conditions under which a message is actually received by its recipients. A message may be delivered at-most-once (i.e., zero or one time), at-least-once (i.e., one or several times), or exactly once. For instance, a reliable broadcast ensures that, if the sender is correct, all correct processes eventually deliver the message exactly once.

Messages in a distributed system are generally not ordered according to the time at which they have been issued (real-time) for several reasons: clocks cannot be perfectly synchronized, messages may be lost and re-sent, the transmission delay of messages and the relative speed of processors in a distributed system are not uniform, etc. These factors demonstrate the need to define a logical ordering of messages (that may differ from real-time ordering).

FIFO (First-In-First-Out) ordering guarantees that messages sent by a single source are delivered to all processes in the same order as they have been produced.

Informally, causal order ensures that a message will not be delivered until all messages on which it depends have been delivered. Dependence is expressed using the *happened before* relationship introduced by Lamport in [10].

Atomic — or total order — broadcast guarantees that all messages sent by different clients are delivered in the same order to each process. There are several variations of the atomic broadcast problem. It may allow incorrect processes to deliver some messages in a wrong order (non-uniform atomic broadcast), or mandate that all processes, correct or not, deliver all messages in the same order (uniform atomic broadcast). It may additionally ensure FIFO or causal order.

Broadcast communication is much used for replication and high availability. It makes it possible to invoke all copies of a replicated service at once and maintain these copies consistent with each other. Each type of broadcast provides different guarantees and has a different cost. An application should use a broadcast that has minimal cost while providing sufficient guarantees to maintain application consistency.

2.2. On Concurrent Execution

Unlike message delivery, which is assumed to be an atomic event, the actual processing of a message can take time. In a client/server interaction, a message from the client to the server typically corresponds to a request, and a message from the server to the client to a reply. The time between the delivery of the request and the generation of the reply corresponds to the request processing time. Message reliability and ordering provides guarantees with respect to message delivery, but not with respect to request processing.

When multiple requests to an object overlap, we say that they execute concurrently. There exist several correctness conditions for concurrent execution, like serializability [14], sequential consistency [2], or linearizability[9]. These correctness conditions specify what happens in the presence of interleaved or overlapping accesses to an object (or to different copies of the same object). They ensure that the concurrent execution of multiple requests are equivalent to some valid non-concurrent execution.

The implementation of an object may be able to execute several requests concurrently and comply with one of the above correctness conditions, for instance by implementing concurrency control using mutual exclusion mechanisms. Other objects may have been implemented without taking concurrency into account and need to have their requests serialized. In that case, the middleware infrastructure needs to wait until the reply of the current message has been produced before delivering the next message to the application. It may also happen that an object supports concurrent execution of some of its operations (e.g., because they modify disjoint parts of the object's state), while the other operations must be serialized.

2.3. From Messages to Requests

Traditionally, a message encapsulates some data agreed upon only by the sender and the receiver. The semantics of messages is unknown outside the scope of the application. For instance, a router or a firewall will not be able to distinguish a read operation from an update.

In a distributed system based on remote procedure calls (RPCs) or remote method invocations (RMIs) however, messages have a specific self-describing format. This format unequivocally identifies the source, the destination, the operation and the arguments of a distributed invocation. Although processes can embed application-specific data in the request's argument, messages must be formatted in an application-independent manner. This property permits the middleware infrastructure to analyze the request and handle it according to its semantics, transparently to the application (e.g., by implementing replication, load-balancing, or caching).

2.4. From Client to Server Choice

Distributed systems that offer several types of communication primitives generally let the client choose the most appropriate primitive for each invocation. For instance, group communication toolkits like Isis [3], Horus [17], Totem [12], or Transis [1] let client applications choose the semantics of each multicast (e.g., reliable, FIFO, total order, or causal order).

However, this scheme has some limitations. It requires the application programmer to know precisely the semantics of each operation and its potential side effects for choosing the most efficient communication primitive. For instance, a read operation may implicitly modify the state of a replicated server (at the implementation level) and thus require a total order communication primitive. The application programmer that develops a client for that server needs to know implementation-specific side effects in order to select the right protocol.

This scheme also introduces the risk of breaking the consistency of the server because of actions performed by a client application (e.g., badly programmed or malicious client). Since clients are generally not controlled by the same entity that hosts the service, this risk must be taken seriously.

To deal with these limitations, one may be tempted to take a conservative approach and use only primitives that provide the highest guarantees, such as atomic broadcast, for all invocations. This approach obviously has a strong impact in terms of performance by increasing latency and decreasing throughput.

A solution to these problems is to shift the responsibility of choosing the right communication protocol from the client application to the middleware infrastructure, which uses semantic knowledge of the server for making decisions. Therefore, the server becomes the entity that actually determines how communication is to be performed. Indeed, the server is the entity that knows best what are its requirements in terms of request delivery, ordering, and concurrency, and that can optimize protocols according to other requests sent or processed concurrently.

2.5. From Specific to Generic Broadcast

Among all the broadcast primitives previously described, one may wonder how to choose the adequate protocol for a specific problem. Obviously, each broadcast type provides a different set of guarantees, and has a specific complexity and overhead. Using a broadcast primitive that guarantees both causal and total order will prove inefficient if the application only requires causal ordering.

Recently, generic algorithms have been proposed [15, 11] to address this problem and implement different ordering guarantees depending on the messages that are broadcast. Informally, these algorithms work by running a sequence of sub-protocols. Each sub-protocol builds on the execution of the previous one and provides stronger guarantees. Protocol execution stops once the guarantees needed for the message are met. This paper does not describe algorithms that implement generic broadcast, but rather motivates the need for such algorithms and describes policies for deciding how to instantiate them.

3. An Example

To illustrate the different semantic properties of an object and their effects on distributed protocols, we consider the example of a simplified bank account. We model a bank account using a Java object, and assume that it is remotely invoked using middleware such as CORBA [13] or Java RMI [16]. The code of the bank account is given in Figure 1.

```java
public class Account {
  private int balance = 0;
  private String name = "<none>";
  private Date start = new Date();

  public String getName()
  { return name; }

  public String setName(String s)
  { name = s; }

  public String getBalance()
  { return balance; }

  synchronized public void deposit(int i)
  { balance += i; }

  synchronized public void withdraw(int i)
    throws OverdraftException
  {
    if(i > balance) throw new OverdraftException();
    balance -= i;
  }

  synchronized public void addInterest()
  {
    Date today = new Date();
    balance += interestForPeriod(start, today);
    start = today;
  }
}
```

Figure 1. Implementation of an bank account.

Now consider that an account object is actively replicated at multiple sites. A quick look at Figure 1 shows that some methods modify the object's state (e.g., "deposit") while other methods don't (e.g., "getName"). In order to keep the replicated copies of the account object consistent, the methods that modify the object's state must be executed on all copies. However, methods that do not modify the object's state can execute on only one copy (or a subset of all copies), thus avoiding unnecessary communication and processing.

Some methods which modify the object's state have the additional property that they can be executed several times with the same effect as if they had been executed only once. For instance, calling "setName" twice with the same parameters has the same effect as calling this operation once. The property is useful for communication protocols that guarantee at least once message delivery but not at most once delivery.

The outcome of some methods can be predicted based on the current object's state and the argument given to the methods. This is the case for instance of "deposit" and "withdraw". When dealing with replication, such methods can be invoked using an active replication policy, where the invocation is sent to and processed by all replicas. However, the outcome of "addInterest" cannot be predicted because it computes the interest using the current time, which is not predictable. Therefore, an invocation to such a method would need to use a passive replication policy, where only one copy processes the request and updates the other copies.

The order in which some methods are executed may not matter. For instance, "getName" and "deposit" return the same results and modify the state of the account in the same manner independent of the order in which they are called. The order in which "deposit" and "withdraw" are executed does however matter. For instance, if the account has a balance of $800 and two invocations are performed: "deposit($300)" and "withdraw($1000)", the withdrawal will succeed only if the deposit has been previously executed. In the context of replication, concurrent invocations do not need to be totally ordered if the order in which the methods execute does not matter.

Finally, some methods can be executed concurrently, either because they use mutual exclusion mechanisms (e.g., "deposit" and "withdraw", which are synchronized) or because they affect disjoint part of the object's state (e.g., "deposit" and "setName"). The invocations to methods that conflict and do not use mutual exclusion mechanisms (e.g., "getBalance" and "deposit") must be serialized.

4. Semantic Properties

This section describes more formally the semantic properties of objects and operations outlined in the previous section, and that can be used to optimize communication in a distributed context. Note that some of these properties are implementation-dependent, i.e., a particular implementation may satisfy these properties while another implementation of the same object may not.

4.1. Read-only Operations

We refer to an operation as *read-only* if it does not modify the state of the target object. Note that an operation with no input parameters is not always read-only (e.g., the "addInterest" operation on the account object does not have input parameters but does modify the object's state). In the example of Figure 1, the methods "getName" and "getBalance" are read-only.

4.2. Idempotent Operations

An operation is *idempotent* if calling the operation twice with the same arguments has the same effect than calling it once. This property is useful because it permits clients to reissue a request when they cannot determine whether the target object has received the request or not. For instance, when an E-Commerce server crashes while processing a purchase request (partial execution), the client can safely reissue the request, if it is idempotent, without risks of being billed twice. A consequence of this definition is that read-only operations are also idempotent. In the example of Figure 1, the methods "getName", "getBalance", and "setName" are idempotent.

4.3. Deterministic Operations

An operation is *deterministic* if its outcome (i.e., the modification to the object's state and the reply sent to the client) depends only on the initial state of the object and on the sequence of operations performed by the object (history). This property is especially useful for replication: if two copies of an object have identical initial states and receive the same sequence of invocations to deterministic operations in the same order, their final state will also be identical. In the example of Figure 1, all methods except "addInterest" are deterministic.

4.4. Commutative Operations

Two operations are *commutative* with respect to each other if the order in which they are called on an object does not matter. In other words, if two copies of an objects have an identical initial state and receive two invocations to commutative (and deterministic) operations in a different order, their final state will also be identical. From this definition, it is trivial to infer that read-only operations are commutative with one another.

Table 4.4 illustrates the commutativity relationships of the methods of Figure 1. In the table, we assume that each operation is scheduled to execute at a specific time, and commuting two operations corresponds to exchanging their scheduled execution time. Thus, "addInterest" is commutative with respect to itself but not to any other operation that modifies the current balance.

	getName	*setName*	*getBalance*	*deposit*	*withdraw*	*addInterest*
getName	Y	N	Y	Y	Y	Y
setName	N	N	Y	Y	Y	Y
getBalance	Y	Y	Y	N	N	N
deposit	Y	Y	N	Y	N	N
withdraw	Y	Y	N	N	N	N
addInterest	Y	Y	N	N	N	Y

Figure 2. Commutativity of methods of the bank account.

4.5. Parallelizable Operations

Two operations of an object are *parallelizable* if their concurrent execution is equivalent to a serial execution of both operations. Operations that modify disjoint parts of an object's state are often parallelizable. Note that two operations that are parallelizable but not commutative need to be executed sequentially on an actively replicated server because the two possible serial executions of both operations produce different results. All the commutative methods of Figure 1 are also parallelizable (see Table 4.4). In addition, non-commutative combinations of "deposit", "withdraw", and "addInterest" are parallelizable.

4.6. Combinable Operations

Two operations of an object are *combinable* if there is an operation whose input can be derived from the inputs of the two aforementioned operations and, if performed in place of the two operations, results in the same outcome as a sequential execution of the two operations. The performance of a distributed systems can be significantly improved by combining multiple consecutive requests and thus reducing the number of invocations. Note that idempotent operations are trivially combinable with themselves. Two operations that modify the same part of an object's state by overwriting previous values can also be combined trivially by discarding the first operation. In the example of Figure 1, "deposit" can be combined with itself by summing arguments, and with "withdraw" by subtracting arguments (when more money is deposited than withdrawn). "setName" is combinable with itself by discarding all but the last invocation.

4.7. Object Properties

When a property described above applies to all the methods of an object, we say that the object has that property. For instance, an object is deterministic if all its methods are deterministic.

There is a special type of object which is of particular interest. An object is *stateless* if it does not maintain a state across multiple invocations. A consequence of this definition is that the operations of stateless objects are read-only, idempotent, and commutative.

5. Towards Intelligent Middleware

This section describes our vision of intelligent middleware and how the semantic properties described in Section 4 can be used to increase the reliability and availability of Internet applications.

5.1. The Challenge

The biggest problem of the Internet nowadays is to be able to cope with the exponentially increasing network traffic. Some Internet web sites and applications must be able to serve thousands of clients simultaneously in near real-time and process hundreds of thousands of requests or transactions per day. Clients are generally widely dispersed in the Internet, making the web site or application server the bottleneck (rather than the network). Mission-critical applications (e.g., banks, stock exchanges, etc.) must provide reliability without impact on performance. This degree of scalability is currently achieved using costly hardware or by customizing and deploying the software on clusters of servers (see Figure 3).

5.2. The Vision

A desirable goal of middleware infrastructures is to make existing Internet services reliable and highly-available. Techniques like replication and load balancing traditionally require programming support from the application, in order to maintain all copies of a server consistent and to determine policies for load balancing (e.g., which requests can be processed by which copy). As a matter of fact, middleware cannot blindly replicate or load balance messages addressed by clients to a replicated service.

We believe that middleware needs two elements for implementing intelligent behavior that transparently strengthen Internet applications:

1. The ability to interpret messages in terms of requests and replies, and to inspect them (in particular, to determine the target object and operation).

2. Some semantic knowledge of the application.

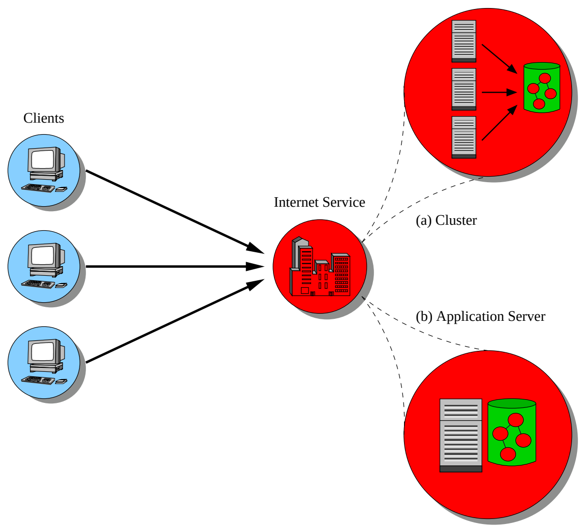

Figure 3. Two approaches to high availability in the Internet: (a) using a cluster of servers and (b) using an application server running on specialized hardware.

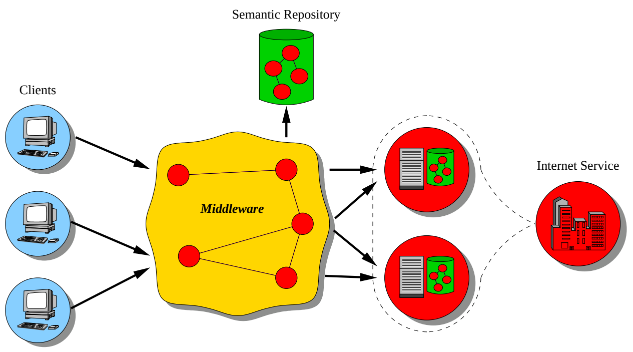

Figure 4. Intelligent middleware makes Internet services reliable and highly available by transparently caching data, load-balancing requests, and maintaining consistency of multiple service instances.

In addition, the application must be deployed according to some guidelines. For instance, to achieve fault-tolerance, multiple instances of the application must be executing concurrently.

When these conditions are met, the middleware infrastructure can understand enough of the application semantics and its interactions with other distributed components to determine when a request must be replicated, when a reply must be cached, when cached data must be invalidated, when a request can be load-balanced, etc. Figure 4 shows the main architecture of such intelligent middleware.

5.3. Specifying, Storing, and Using Semantic Knowledge

Although the semantic description of a service generally depends on its implementation, that description does not need to be specified in the program by the developer. Semantic information about the server's behavior can be specified at deployment time. We propose that this information be stored in a *semantic repository*. The middleware infrastructure accesses this information at runtime and uses it to mediate interactions between clients and servers in an optimal manner. No action is needed from the application other than the initial specification of the server's properties. Thus, our middleware infrastructure can be used with "legacy" applications to make them highly available and reliable. We realize that having the middleware check the semantics of each client request will add some overhead to the dispatching of requests, but we believe that the advantages, some of which we present in the next section, will outweigh the disadvantage of the extra processing time.

Note that we only discuss here the static properties of a distributed object. Dynamic properties that depend on both the request and the current state of an object could be another step towards intelligent middleware.

Semantic knowledge of distributed objects can be used for several tasks. In the rest of this section, we describe four domains where this knowledge can significantly improve the reliability and availability of distributed applications.

Message Protocols. Message-level protocols are probably the area where the semantic knowledge of an object is most beneficial. When using a point-to-point communication protocol, an idempotent operation allows the middleware infrastructure to deliver a message more than once without breaking the consistency of the target object. In other words, exactly-once semantics can be implemented using a protocol that guarantees at-least-once semantics.

In a replicated environment, a read-only operation can be directed to only one copy (e.g., the closest, the fastest, or the least-recently accessed) or a small subset of all copies in order to avoid unnecessary redundant processing. A *multicast* can thus be replaced transparently by an *anycast*.

Operation commutativity is especially useful with active replication. When several clients issue concurrent invocations to commutative operations, the communication protocol does not need to totally order these invocations, thus significantly reducing their overhead. The protocols of [15] and [11] are examples of protocols that implement both reliable or total-order broadcast depending on a dynamic conflict relation. These protocols can directly benefit from the knowledge of commutativity between operations.

Replication Protocols. Fault-tolerance in distributed systems is typically achieved through replication. The literature distinguishes between two main classes of replication techniques: passive replication and active replication [8]. In passive replication the client only interacts with one replica, called the *primary*: the primary handles the client request and sends back the response. The primary also issues messages to the secondaries (the other replicas) in order to update their state. In active replication the client sends its request to all the replicas, which all handle the request and send back the response to the client. The client waits only for the first reply. Note that active replication requires the servers to be deterministic.

There is a tradeoff between the two replication techniques. Active replication provides a more predictable and generally faster response time than passive replication, and requires less application support. On the other hand, passive replication is better adapted when processing a request is time-consuming and/or when update messages are significantly less expensive than messages exchanged between clients and servers.

While the replication policy is generally specified at the process or object level, some replication frameworks make it possible to define that policy on a per-invocation basis [5]. In such environments, the semantic knowledge of an object can help in choosing a replication policy. The middleware infrastructure will use passive replication for all non-deterministic methods. Active replication may be preferred for deterministic methods, unless specified otherwise by the application deployer.

Request Execution. When a server executes a time-consuming request (e.g., a database query), it may be desirable to allow other requests to execute concurrently. However, this requires the server to be programmed accordingly (e.g., in a multi-thread safe manner). For servers that do not support concurrent operations, requests must be processed sequentially by delaying delivery of a request until processing of the previous one has completed.

Even if an object has not been programmed with concurrency in mind, it may happen that some of its requests are parallelizable (e.g., because they affect disjoint part of the object's state). It is thus useful to specify which requests can be executed concurrently as part of the object's semantic description, and take advantage of that information to optimize request delivery.

Reply Caching. When a client accesses a read-only deterministic request, it may be advantageous to cache the reply in the middleware infrastructure to speed up subsequent invocations to the same operation. The cache is maintained up-to-date on behalf of the application, and is invalidated when a conflicting operation is invoked on the object. A general rule is that the cache needs to be invalidated upon invocation of an operation that does not commute with the operation of the cached reply.

6. Conclusion

In this paper, we have discussed how the knowledge of the semantics of distributed objects can help in implementing intelligent middleware that increases the reliability and availability of Internet applications in a near-transparent manner. We have presented different semantic properties that can help distributed middleware to take decisions on behalf of the application and choose optimal protocols for remote object invocations. The semantic properties presented here is my no means and exhaustive list, and we exploring other properties that we think can be useful. Some generic distributed algorithms [15, 11] are already adapted to take advantage of such semantic knowledge.

We believe that numerous Internet applications would benefit from middleware able to understand major distributed interaction protocols (e.g., IIOP, RMI, HTTP) and provide instant reliability by implementing transparent replication, load balancing, and caching. This approach has the key advantage to implement reliability and high availability inside the middleware infrastructure, instead of having to have them directly programmed in the applications.

We have successfully implemented an HTTP gateway that transparently manages reliability and availability of web-based services, using a combination of replication and load-balancing. The gateway uses semantic knowledge to discriminate between requests to static data (e.g., static web pages), read requests to session data (e.g., contents of a virtual shopping cart), and requests that modify session data. The applicability of this gateway is restricted to HTTP requests and only makes limited use of semantic knowledge. We are currently in the process of building a more comprehensive object-based infrastructure to evaluate the effectives of the ideas presented in this paper. It is especially important to see how the additional overhead of checking the semantics of requests compares to the gains made by using this knowledge.

References

[1] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication sub-system for high availability. In *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing (FTCS-22)*, pages 76–84, July 1992.

[2] H. Attiya and J. Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems*, 12(2):91–122, May 1994.

[3] K. Birman and R. Van Renesse. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, 1993.

[4] A. Farrag and M. Özsu. Using semantic knowledge of transactions to increase concurrency. *ACM Transactions on Database Systems*, 14(4):503–525, Dec. 1989.

[5] P. Felber, X. Défago, P. Eugster, and A. Schiper. Replicating corba objects: A marriage between active and passive replication. In *Proceedings of the 2nd IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems (DAIS'99)*, Helsinki, Finland, June 1999.

[6] P. Felber, R. Guerraoui, and A. Schiper. The implementation of a CORBA object group service. *Theory and Practice of Object Systems*, 4(2):93–105, 1998.

[7] H. Garcia-Molina. Using semantic knowledge for transaction processing in distributed database. *ACM Transactions on Database Systems*, 8(2):186–213, Mar. 1983.

[8] R. Guerraoui and A. Schiper. Software-Based Replication for Fault Tolerance. *IEEE Computer*, 30(4):68–74, April 1997.

[9] M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, July 1990.

[10] L. Lamport. Time, clocks and ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[11] H. F. M.K. Aguilera, C. Delporte-Gallet and S. Toueg. Thrifty generic broadcast. In *Proceedings of the 14th International Symposium on Distributed Computing (DISC'00)*, Oct. 2000.

[12] L. Moser, P. Melliar-Smith, D. Agarwal, R. Budhia, and C. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, Apr. 1996.

[13] OMG. *The Common Object Request Broker: Architecture and Specification*. OMG, October 1999.

[14] C. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, Oct. 1979.

[15] F. Pedone and A. Schiper. Generic broadcast. In *Proceedings of the 13th International Symposium on Distributed Computing (DISC'99)*, Sept. 1999.

[16] Sun. *Java Remote Method Invocation - Distributed Computing for Java (White Paper)*, 1999.

[17] R. van Renesse, K. Birman, and S. Maffeis. Horus, a flexible group communication system. *Communications of the ACM*, 39(4), April 1996.