On Bootstrapping Replicated CORBA Applications *

W. Zhao, L. E. Moser and P. M. Melliar-Smith
Department of Electrical and Computer Engineering
University of California, Santa Barbara, CA 93106
wenbing@alpha.ece.ucsb.edu, moser@ece.ucsb.edu, pmms@ece.ucsb.edu

Abstract

Critical components of a distributed system must be replicated to achieve high availability and fault tolerance. Current fault-tolerant CORBA infrastructures have concentrated on mechanisms for object replication and recovery, while rarely considering practical issues related to the context, i.e., the CORBA middleware within the process in which the object runs. Our study shows that to replicate and recover complex CORBA applications, the behavior of the process that hosts the CORBA objects, in particular, the bootstrapping of an application, must be taken into account. In this paper, we discuss the challenges that arose when bootstrapping CORBA applications in some common scenarios, and we provide strategies to handle such difficulties so that CORBA applications can be rendered fault-tolerant.

1 Introduction

The Common Object Request Broker Architecture (CORBA) [13] is one of the most general distributed computing architectures. Applications based on CORBA can benefit greatly from its openness, its use of object technology, and its support for interoperability and portability. Efforts to enhance CORBA with fault tolerance have resulted in a number of sophisticated fault-tolerant CORBA infrastructures, including AQuA [2], OGS [3], IRL [6], Eternal [10], Doors [11] and PluggableFT [15].

Most of these fault-tolerant CORBA infrastructures have focused on the development of mechanisms for *object* replication and recovery. A state-machine approach has been widely used in these systems. The object being replicated is regarded as a state machine such that:

- The object state is unchanged if there is no input (in CORBA, the input takes the form of a remote method invocation on the object).
- Given the same input, the current object state transforms to the same next state.

*This research has been supported by DARPA/ONR Contract N66001-00-1-8931 and MURI/AFOSR Contract F49620-00-1-0330.

In such fault-tolerant CORBA systems, it is implicitly or explicitly assumed that the application is a collection of state machines formed by the CORBA objects and, hence, the application can also be treated as a state machine. While objects are first-class citizens in the CORBA world, they do not exist alone by themselves. The objects are created, managed, and deleted in the scope of Portable Object Adaptors (POAs). The POAs, in turn, exist in the scope of Object Request Brokers (ORBs). One or more ORBs are created and run in the scope of a process, *i.e.*, the run-time image of an application in an operating system. Before an object can provide service to other objects or clients, *i.e.*, the object starts to run as a state machine, the process that hosts them must have finished the bootstrapping stage. By bootstrapping, we mean the operations and actions taken by an application after it is started as a process and before the control of the main thread of the process is handed over to the ORB. Thus, when replicating and recovering CORBA applications, we cannot consider only the operations and the state of the CORBA objects, we must also take into account the POA, the ORB and the bootstrapping operations of the process that hosts the objects.

In practice, it is fairly common that an application communicates with other remote objects or applications in its bootstrapping stage, for various reasons:

- The application needs to register the references of some of its CORBA objects with other CORBA Common Object Services (COS) [12], such as the COS Naming Service, so that other objects can locate and invoke methods of these objects, or the COS Notification Service, so that they can receive callbacks for the events that occurred.
- The application needs to retrieve, from the COS Naming Service or the COS Trading Service, the references of the objects that it will later invoke.
- The application narrows a remote object reference to a derived type of the type specified in the `type_id` field of the object IOR. An `_is_a()` method invocation is sent to the remote object for additional information.

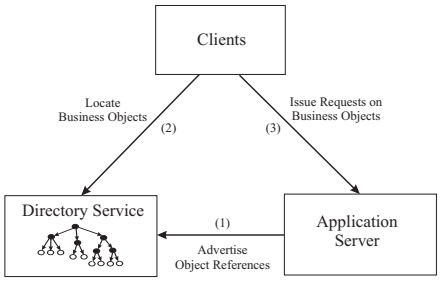

Figure 1. An example of an application that uses the COS Naming Service as a directory service. The bootstrapping of the application server is not silent.

- The third-party library linked into the application might wish to perform some bootstrapping operations with other remote services. For example, in an application that involves a distributed transaction using the COS Object Transaction Service and the Oracle Database Management System (DBMS), the Oracle client-side library linked into the application engages a number of rounds of dialog with the Oracle DBMS for recovery purposes.

An example of such an application that uses the COS Naming Service as a directory service is shown in Figure 1. The presence of the remote communication during the bootstrapping stage of an application causes a deviation from the state-machine model (because the state of the application is changed without an input) and, therefore, is largely unexpected by the replication and recovery mechanisms in existing fault-tolerant CORBA infrastructures. This introduces the following challenges:

- *Handling Duplicate Messages.* With replication of an application, each incoming message to a replica must be examined by the replication mechanisms to see if it is a duplicate. A typical technique to achieve duplicate detection is to carry an operation identifier [9] with each message. The replication mechanisms at the receiving end compare the operation identifier with the expected one and discard a message if it is deemed to be a duplicate.

  This scheme works fine under the state-machine model; however, it prevents applications from adding a new replica to an existing group or recovering a failed replica if remote operations show up during the bootstrapping stage, because a late starting replica or a recovering replica's initial requests are regarded as duplicates and the replica is blocked forever waiting for replies that will never arrive.

- *Synchronizing State Transfer.* The recovery of a replica requires that the application state is transferred from an existing replica to the recovering replica. Usually, incoming messages to a recovering replica are queued until the application state has been restored. However, if remote communication occurs during the bootstrapping of an application, this prevents a replica from recovering even without the duplicate suppression problem because the replies are queued in the recovery log, rather than delivered to the recovering replica.

In this paper we investigate and compare two strategies for handling the unexpected communication during the bootstrapping of an application: client-side logging and server-side logging. We have implemented the related mechanisms in our recently developed Pluggable FT infrastructure [15].

2 Related Work

Among existing fault-tolerant CORBA infrastructures [1, 2, 3, 5, 6, 8, 9, 11], there are very few that have considered the various problems introduced by the middleware and the application process on the replication and recovery mechanisms other than application object state.

In Eternal [9], Narasimhan *et al.* pointed out that the state of the ORB/POA that hosts the application objects must be transferred together with the state of the application object during recovery to achieve strong replica consistency. However, the issues of replication and recovery in the presence of remote invocations during the bootstrapping of an application were not considered. However, both Eternal [9] and AQUA [2] have considered the state introduced by the fault tolerance infrastructure itself.

Recognizing the limitations of existing approaches for fault-tolerant CORBA systems, we recently developed a Pluggable Fault Tolerant CORBA infrastructure (PluggableFT) that follows a novel non-intrusive integrated approach [15]. In PluggableFT, the fault tolerance mechanisms are embedded into the ORB address space using the pluggable protocols framework that is common to most modern ORBs [4].

This approach provides easy access to the middleware-level state that is necessary to recover a replica consistently. It also makes it possible for the replication mechanisms to collaborate with the application logic to achieve fault tolerance. Indeed, the solutions that we present in this paper require such collaboration so that the fault tolerance mechanisms can distinguish between the remote invocations during bootstrapping and the invocations thereafter.

3 Handling Bootstrapping Operations

The applications that engage in remote operations during bootstrapping notify the fault tolerance mechanisms of the

completion of such operations by invoking an Application Programming Interface (API) provided by the fault tolerance infrastructure. In this way, the fault tolerance mechanisms can distinguish the messages within the scope of bootstrapping from those outside.

We refer to the requests issued by a replica before the completion of bootstrapping as *bootstrapping requests*, and the corresponding replies as *bootstrapping replies*. *Bootstrapping messages* consist of both bootstrapping requests and bootstrapping replies. Likewise, we refer to (nested) requests issued by a replica after the completion of the bootstrapping, and the requests from a pure client, as *normal requests*. We refer to the corresponding replies as *normal replies*. *Normal messages* consist of normal requests and normal replies.

We have developed two alternative strategies to handle remote operations during the bootstrapping stage. The first strategy involves server-side logging of the bootstrapping messages. The advantage of this strategy is that the state transfer mechanisms are almost identical to that which supports well-behaved applications (*i.e.*, those that can be regarded as a composite state machine). Therefore, minimum changes are needed to adapt the existing fault tolerance mechanisms to support applications with remote bootstrapping operations. However, server-side logging pushes most of the responsibility of handling such operations to mechanisms that support other service providers. The recovery of such service provider replicas might need to transfer large logs for the bootstrapping replies.

The second strategy involves client-side logging of bootstrapping messages. This strategy is more sophisticated and relieves the service provider mechanisms from the burden of handling application bootstrapping operations and enables faster recovery for replicas of both the service providers (because they no longer need to transfer potentially large bootstrapping message logs on the applications behalf) and the application servers that use these services (because the transmission of one large message is more efficient than the transmission of many small messages.

3.1 Server-Side Logging

The bootstrapping requests from a replica must be handled differently from normal requests. This difference is carried by the FT protocol header that is transmitted with each message. The request information is stored in the server's transport instance so that the reply message of a non-duplicate bootstrapping request can be recognized as a bootstrapping reply message. At the server-side, the replication mechanisms store all bootstrapping replies in the Bootstrapping (BS) Reply Log, as shown in Figure 2. In step 1, a bootstrapping request is retrieved from the message log and is delivered to the replica. When the bootstrapping reply is ready, in step 2, it is copied to the BS Reply Log, and in step 3, the message is multicast.

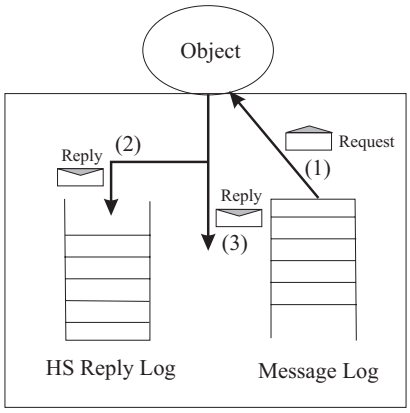

Figure 2. Server-side logging of the bootstrapping replies. In the diagram, the bootstrapping flag in a message is illustrated by the shaded envelope flap.

Duplicate Message Handling. The duplicate message handling is shown in Figure 3. Non-duplicate messages are passed to the Message Log for future delivery (Figure 3a). When the replication mechanisms receive a duplicate bootstrapping request, they find the corresponding bootstrapping reply in the log and return the message (Figure 3b). They discard an incoming normal message if it is a duplicate without further processing (Figure 3c).

Synchronizing State Transfer. The replication mechanisms send all bootstrapping requests issued by a recovering replica, and deliver the corresponding bootstrapping replies as soon as they receive them. The state transfer SET_STATE message, which the fault tolerance mechanisms might receive before completion of the bootstrapping, is delayed until the bootstrapping is complete. The fault tolerance mechanisms queue other incoming messages after the start of recovery and before the SET_STATE message.

To recover a server replica, we transfer the bootstrapping reply log, as part of the infrastructure state, from the existing replica to the recovering replica.

3.2 Client-Side Logging

Alternatively, the bootstrapping reply messages can be logged at the client-side, *i.e.*, by the fault tolerance mechanisms supporting the replicas that issued such bootstrapping requests, as shown in Figure 4. This strategy requires more complicated mechanisms for state synchronization, but provides a number of advantages.

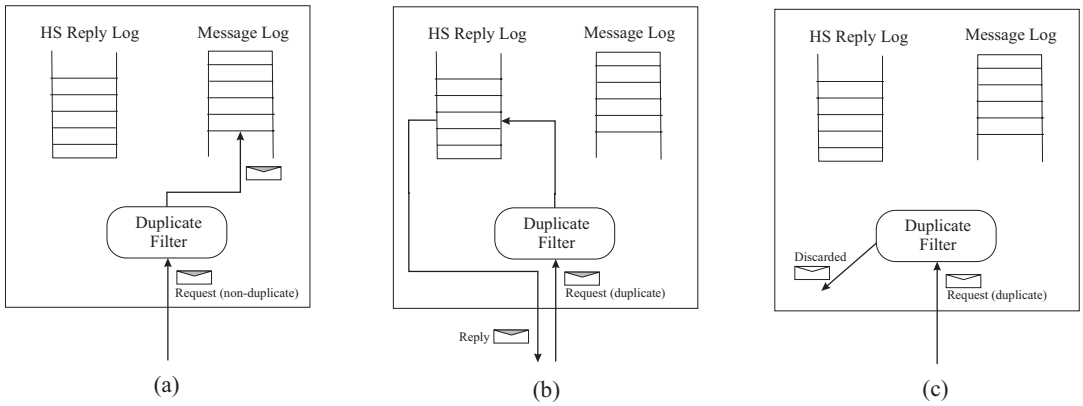

Figure 3. Duplicate detection for incoming messages. In the diagram, an envelope with a shaded flap stands for a bootstrapping message; an envelope with an unshaded flap stands for a normal message.

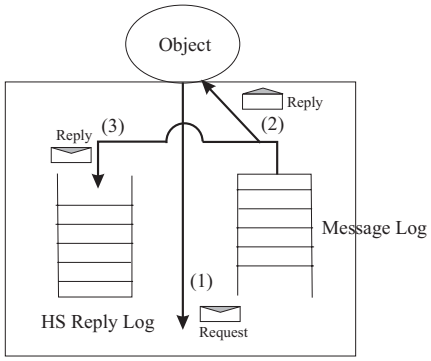

Figure 4. Bootstrapping reply messages are logged by the fault-tolerance mechanisms immediately after they are delivered to the local replica at the client.

The bootstrapping reply messages are logged initially at the first replica in a group. Part, or all, of the BS Reply Log is transferred to the late starting replicas, depending on the delay between the first and later started replicas. Unlike server-side logging, client-side logging does not require the server-side fault tolerance mechanisms to carry any additional responsibility. On the server-side, filtering of duplicates can be performed as usual.

The steps of recovering a replica when the existing replicas have completed the bootstrapping operations are shown in Figure 5.

Step i. The fault tolerance mechanisms detect the join of a new replica and broadcast a GET_STATE protocol message to indicate the start of the recovery operations.

Step ii. The recovering replica might actually wish to send the first bootstrapping request message before, or after, the receipt of the GET_STATE message. In either case, the request is recognized as a bootstrapping message and is discarded. Because there is a pending reply, the recovering replica is blocked from issuing further bootstrapping requests.

Step iii. Normal requests from clients of the replicated server might arrive after the GET_STATE message. These requests are appended to the message log after the GET_STATE message. Although not shown in the diagram, normal reply messages for a normal request (or nested request) that the local replica sent might also arrive. Again, they are appended to the message logs at both the existing replica and the recovering replica.

Step iv. After the fault tolerance mechanisms at the existing replica have delivered all of the messages in the message log and have reached an operational quiescent state, the mechanisms start retrieving application object state by invoking the `get_state()` method of the object. All objects being replicated are required to inherit the Checkpointable interface containing the `get_state()` and `set_state()` methods and to register the object reference with the fault tolerance infrastructure once it is activated.

Step v. The retrieval of the application state is a simple local method invocation. The returned state, together with the BS Reply Log and other middleware and infrastructure state information, are transferred as the payload of the SET_STATE protocol message from the existing replica to the recovering replica. After the receipt of the SET_STATE message, the mechanisms at the recovering replica extract the BS Reply Log and copy it to the local BS Reply Log. The remainder of the SET_STATE message is put at the top of the message log (replacing the GET_STATE message). There

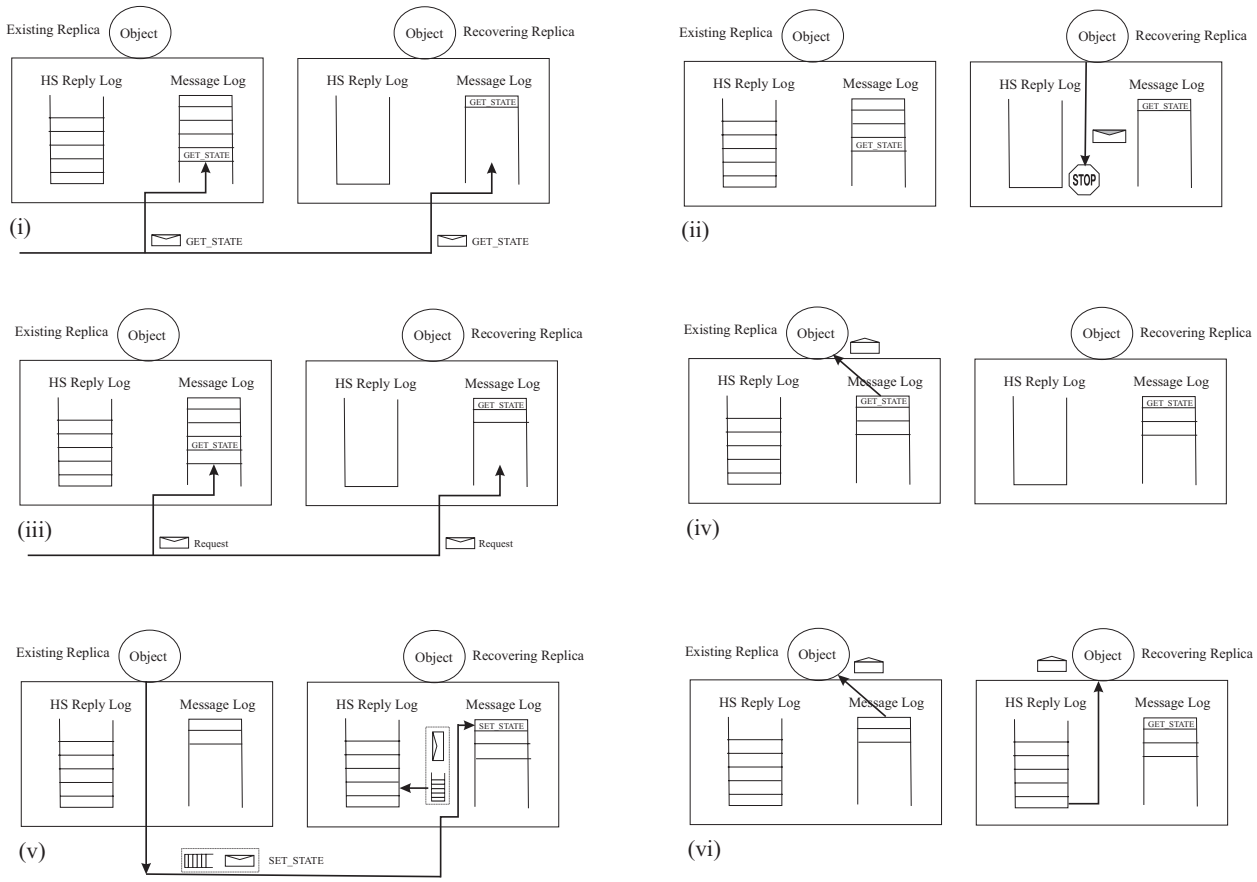

Figure 5. State synchronization for client-side logging.

is no further processing of the SET_STATE message until the recovering replica finishes the bootstrapping operation.

Step vi. The mechanisms at the existing replica start delivering the next request in the message log. At the recovering replica, the mechanisms locate in the BS Reply Log the corresponding reply for the first bootstrapping request (if the replica has issued such a request) and deliver the message to the local replica. Bootstrapping reply messages are retrieved from the BS Reply Log for all subsequent bootstrapping requests. Once the bootstrapping operations are completed, the mechanisms apply the application state to the recovering object, again using a local method invocation. Other middleware and infrastructure state is also set at this point.

The synchronization of state transfer for a late starting replica, while the first replica of the object is still busy with the bootstrapping operations, is slightly different. In Step (iv), when the mechanisms at the existing replica are ready to process the GET_STATE message, no application state is retrieved. Only the BS Reply Log is prepared as the payload of the SET_STATE message for the recovering replica, because application state can be reconstructed at the recovering replica by replaying the bootstrapping replies. In Step (v), the BS Reply Log is transferred in the SET_STATE message to the recovering replica. At the recovering replica, its local BS Reply Log is populated using the log contained in the payload of the SET_STATE message. The previous GET_STATE message and the SET_STATE message are subsequently discarded. At Step (vi), normal operation resumes at both the existing replica and the recovering replica. Note that, when a reply is expected for a bootstrapping request, both the BS Reply Log and the normal message log are searched. If the reply is found in either of the logs, the reply is delivered to the application and the bootstrapping request is suppressed. The reply is appended to the BS Reply Log if it is not already present in the log. If the reply is not found, the bootstrapping request is sent to its destination group for processing.

4 Performance

We have investigated the effectiveness of the two strategies for handling remote operations during application bootstrapping in our fault tolerance infrastructure.

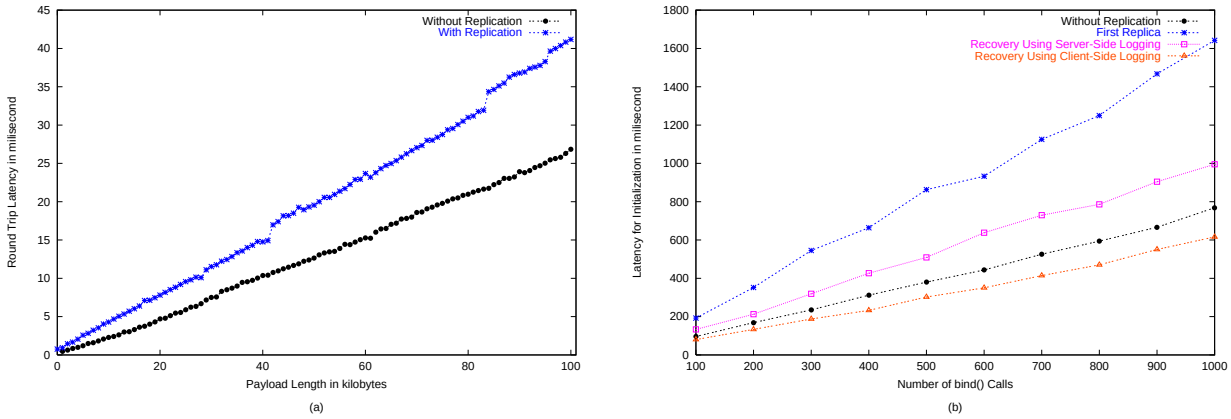

Figure 6. Latency measurement results. (a) Round-trip latency as a function of payload length, without and with replication. (b) Bootstrapping latency as a function of the number of the bind operations, without and with replication, where either server-side or client-side logging is used.

4.1 Experimental Setup

Our experimental testbed consists of four Pentium III PCs, each with a 1GHz CPU, 256MBytes of RAM, running the Mandrake Linux 7.2 operating system, over a 100Mbit/s Ethernet. There was no other traffic on the network.

Our test application is similar to that depicted in Figure 1. It is programmed using ORBacus 4.0 for C++ [14] from Object Oriented Concepts (now Iona), and the ORBacus Naming Service is used as the directory service. The application server creates a number of CORBA objects, activates them and binds (advertizes) them to the Naming Service, before the ORB event loop is started. The number of objects created and, therefore, the number of bind operations, is configured from command-line arguments. The client retrieves the object references of the application server's advertized objects from the Naming Service and then invokes methods of those objects. All of the objects provide an echo method for clients to invoke. A client submits a sequence of longs, and the server returns an identical sequence to the client. There is no substantial computation within the echo method.

In our experiments, both the Naming Service and the application server are three-way actively replicated, and the client is unreplicated. The run-time cost of the fault tolerance infrastructure is determined by comparing the average (1000 times) end-to-end latency of the echo remote method invocation with various lengths of payload, with and without replication. Because the fault tolerance infrastructure does not interpret the parameter types, the cost of replication depends directly on the message length of the payload of the remote method invocation, and not on the parameter types. To focus on the cost of the bootstrapping operations between the application server replica and the directory service during recovery, we assigned each object a fixed length of 1Byte state. The number of bind operations during application server bootstrapping varies between 100 to 1000. The total recovery time, including bootstrapping, is the interval between entering the main() function and completing the bootstrapping operations.

4.2 Results

The end-to-end latency of a synchronous remote invocation as a function of the payload in request and reply message is plotted in Figure 6(a). Not surprising, it is more effective to send one big message (for a payload of 100KBytes, the cost of replication is about 50%) than to send many small messages (for a 1KB payload, the cost of replication is about 100%). The small jumps in latency around 40KBytes and 80KBytes payload in the replicated case are due to multicast protocol's flow control mechanisms [7].

Because the bind request and reply messages are very small (all less than 1KByte), there is high replication overhead (110% - 130%) for the first replica of the application server to do the bootstrapping operations with the Naming Service. Using server-side logging, the recovery time for the other replicas of the application server (assuming the first replica has finished the bootstrapping) is about 30% to 40% longer than the time needed to bootstrap an unreplicated application server. The replication cost is smaller for the late starting or restarted replica than for the first replica because the reply message is retrieved from the log and is sent to the recovering replica without any processing by the ORB and the Naming Service. Using client-side logging, we see a much shorter recovery time than for server-side logging, as a result of aggregating all of the bootstrapping

replies in a single message. The recovery time using client-side logging is less than the bootstrapping time of an unreplicated application server by as much as 20%.

5 Conclusion

Application servers are usually programmed in a flexible manner to accommodate practical requirements; therefore, they often deviate from the state-machine model that has been used by fault tolerance infrastructures. Deviation from the state-machine model introduces a number of challenges for replicating and recovering such applications. In this paper we have focused on the issues related to the presence of remote invocations during the bootstrapping of application servers. We have provided mechanisms for consistent replication and recovery using two alternative strategies for handling bootstrapping reply messages, namely, server-side logging and client-side logging. Our experiments show that client-side logging is more advantageous, albeit more sophisticated, than server-side logging. The recovery of an application server replica for client-side logging is much faster than the recovery of an application server replica for server-side logging. For client-side logging, the recovery time is as much as 20% less than the bootstrapping time of a non-replicated application server.

References

[1] K. P. Birman and R. van Rennesse. *Reliable Distributed Computing Using the Isis Toolkit*. IEEE Computer Society Press, 1994.

[2] M. Cukier, J. Ren, C. Sabnis, W. H. Sanders, D. E. Bakken, M. E. Berman, D. A. Karr, and R. Schantz. AQuA: An adaptive architecture that provides dependable distributed objects. In *Proceedings of the IEEE 17th Symposium on Reliable Distributed Systems*, pages 245–253, West Lafayette, IN, October 1998.

[3] 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.

[4] F. Kuhns, C. O'Ryan, D. C. Schmidt, O. Othman, and J. Parsons. The design and performance of a Pluggable Protocols Framework for Object Request Broker middleware. In *Proceedings of the IFIP Sixth International Workshop on Protocols For High-Speed Networks*, pages 81–98, Salem, MA, August 1999.

[5] S. Landis and S. Maffeis. Building reliable distributed systems with CORBA. *Theory and Practice of Object Systems*, 3(1):31–43, 1997.

[6] C. Marchetti, M. Mecella, A. Virgillito, and R. Baldoni. An interoperable replication logic for CORBA systems. In *Proceedings of the International Symposium on Distributed Objects and Applications*, pages 7–16, Antwerp, Belgium, September 2000.

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

[8] L. E. Moser, P. M. Melliar-Smith, and P. Narasimhan. Consistent object replication in the Eternal system. *Theory and Practice of Object Systems*, 4(2):81–92, 1998.

[9] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. State synchronization and recovery for strongly consistent replicated CORBA objects. In *Proceedings of the IEEE International Cnference on Dependable Systems and Networks*, pages 261–270, Goteborg, Sweden, July 2001.

[10] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Strong replica consistency for fault-tolerant CORBA applications. In *Proceedings of the IEEE 6th Workshop on Object-Oriented Real-Time Dependable Systems*, pages 10–17, Rome, Italy, January 2001.

[11] B. Natarajan, A. Gokhale, S. Yajnik, and D. C. Schmidt. DOORS: Towards high-performance fault-tolerant CORBA. In *Proceedings of the International Symposium on Distributed Objects and Applications*, pages 39–48, Antwerp, Belgium, September 2000.

[12] Object Management Group. The Common Object Services specification. OMG Technical Committee Document formal/98-07-05, July 1998.

[13] Object Management Group. The Common Object Request Broker: Architecture and specification, 2.4 edition. OMG Technical Committee Document formal/2001-02-33, February 2001.

[14] Object-Oriented Concepts, Inc. *ORBacus for C++ and Java*, 2001.

[15] W. Zhao, L. E. Moser and P. M. Melliar-Smith. Design and implementation of a pluggable fault tolerant CORBA infrastructure. In *Proceedings of the International Parallel and Distributed Processing Symposium*, Fort Lauderdale, FL, April 2002.