Performance Consequences of Inconsistent Client-side Membership Information in the Open Group Model

Hein Meling
meling@acm.org

Dept. of Electrical and Computer Engineering
Stavanger University College, Norway

Bjarne E. Helvik
bjarne@item.ntnu.no

Dept. of Telematics, Q2S Centre
Norwegian University of Science and Technology

Abstract

In a distributed fault-tolerant server system realized according to the open group model, inconsistency will (temporarily) arise between the dynamic membership of the replicated service and its client-side representation in the event of server failures and recoveries. The paper proposes techniques for maintaining this consistency and discuss their performance implications in failure/recovery scenarios where clients load balance requests on the servers. Comparative performance measurements is carried out for two of the proposed techniques. The results indicate that the performance impact of lacking consistency is easily kept small, and that the cost of the technique is small.

1 Introduction

The increasing use of online services in our day-to-day activities has mandated that the provided services remain *available* and that they perform their operations *correctly*. To accommodate these goals, it is common to *replicate* critical system components (servers) in the form of *object groups* [3]. Consistency among the object group members is typically guaranteed through a *group communication service* [1]. In the *open* group model [2], external clients interact transparently with the object group, as if it were a single, non-replicated server object. This is different from the *closed* group model, in which clients must become member of the group prior to any interaction with it, thus making it less scalable with respect to the number of simultaneous clients. In order for clients to communicate with the object group, they need to obtain an object group reference. Typically this is accomplished using a naming/registry service [7]. The task of a naming service is to map a textual service name to a remote object reference, also called stub or proxy. Such a proxy is used by clients to access a remote service object. For a replicated service it is common that this (client-side) proxy hold information about the entire object group, allowing the client-side to perform failover to a different group member should some member have failed.

This paper addresses issues concerning maintaining consistency between the dynamic server-side group membership and the client-side proxy representation of that membership. Unless the client-side membership is updated in some way, the client will become exposed to server-side failures. Another aspect is keeping the naming service consistent with the server-side group membership. Building a dependable distributed middleware platform requires the naming service to be fault tolerant. Several existing middleware platforms provide a dependable naming service, including Jgroup [7] and Aroma [9]. However, none of these naming services update their database of client-side proxies in the presence of failures. The proxy will contain also references to failed servers, forcing the client to perform failover for the same server multiple times, leading to increased failover latency.

This paper presents an extension to the client-side proxy mechanism of the Jgroup/ARM [8, 6] object group middleware platform and its dependable naming service [7], to maintain consistency between the server-side group membership and its representations both at the client-side and in the naming service database. The paper is structured as follows. Section 2 gives an overview of the Jgroup/ARM replication management framework, and the dependable registry service. Section 3 discuss client-side performance impairments, and various times involved in the update problem. Section 4 presents the suggested leasing and notification techniques for maintaining consistency with the dependable registry. In Section 5 several solutions to the client-side update problem is discussed, and in Section 6 we provide measurement results and a comparative evaluation of two client-side update techniques. Section 7 concludes the paper.

2 Jgroup/ARM Overview

Jgroup is a novel object group-based middleware platform that integrates the Java RMI and Jini distributed object models with object group technology and includes numerous innovative features that make it suitable for developing modern network applications. For a description of its capabilities, see [8].

ARM [4, 6] is a replicated dependability manager that is built on top of Jgroup and augments it with mechanisms for the automatic management of complex applications based on object groups. ARM handles both replica distribution, according to an extensible *distribution policy*, as well as replica recovery, based on a *replication policy*. Both policies are group-specific, and this allows the creation of object groups with varying dependability requirements and recovery needs. A mechanism is provided to collect and interpret failure notifications from the underlying group communication system. This information is used to trigger group-specific recovery actions in order to reestablish system dependability properties after failures.

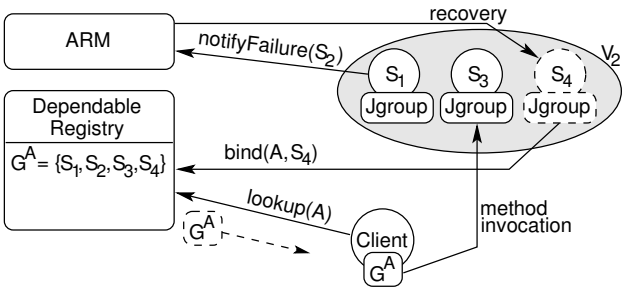

Figure 1: Jgroup/ARM overview.

Figure 1 shows a simplified overview of the workings of Jgroup/ARM. Initially, a group of three members install a view $V_0 = \{S_1, S_2, S_3\}$, and after some unspecified time S_2 crashes. This leads to the installation of a new view, $V_1 = \{S_1, S_3\}$, after which S_1 (the leader) notifies ARM of the failure. ARM will attempt to maintain the redundancy level by installing a replacement replica, S_4, which leads to installation of view $V_2 = \{S_1, S_3, S_4\}$. During initialization of S_4, bind() is invoked on the registry in order to associate the S_4 group member with the G^A client-side proxy. Later, once a client needs to access group A, it contacts the registry to obtain the client-side proxy, G^A. Given G^A, the client can perform invocations on all live members of the group. Note that S_2 still remains in the registry database, even though it has crashed. This is since there is no update mechanism in place.

Several time values related to the failure scenario just described, are shown in the timeline in Figure 3. Let t_0 denote the time at which S_2 crashes. Let t_{V_1} be the time that it takes for the other servers to detect and agree on a new view (V_1), while t_{V_2} is the time it takes to install a replacement server (S_4) and for the servers to agree on a new view (V_2). Thus, the sum corresponds to the recovery time, $t_r = t_{V_1} + t_{V_2}$.

Jgroup includes a *dependable registry service* [7, 8], that can be used by servers to advertise their ability to provide a given service identified by a service name. In addition, clients will be able to retrieve a group proxy for a given service, allowing them to perform method invocations on the group of servers providing that service. The dependable registry service is in essence an actively replicated database, preventing the registry from becoming a single point of failure. The database maintains mappings from service name to the set of servers providing that particular service. Thus, each entry in the database can be depicted as follows: $Name \rightarrow \{S_1, S_2, \ldots, S_n\}$, were S denotes a server, and n represents the number of servers registered under $Name$ in the registry.

3 Client-side Performance Issues

3.1 Performance Without Updates of the Client-side Proxy

To demonstrate the performance penalty of not updating the client-side membership in accordance with the dynamic server-side group membership, we performed several experiments on a four server system with crashes and recoveries in which the clients did not update their membership. The clients perform load balanced invocations on all known servers. The method invoked takes a 1000 byte array as argument, and returns the same array back to the client.

Figure 2 shows the results of the experiment. The plot shows several lines for various client loads ranging from 7 to 70 clients. Only 7 physical machines are used for the clients, whereas in *all cases* the four servers run on dedicated machines, i.e., only the number of live servers remaining in the client-side proxy varies. Initially, all client-side proxies contained all four servers. During the experiment servers crashed and recovered, rendering the client-side proxies inconsistent (having fewer live servers). Not surprisingly the results show that the client-side proxies should update their membership information to avoid increased invocation latencies. Once a client detect a server failure, it is removed from the client-side proxy. Thus, the observed performance drop is due to contention at the servers. Another, perhaps more important problem is the scenario were all servers crash before updating the proxy, rendering failed servers visible to clients.

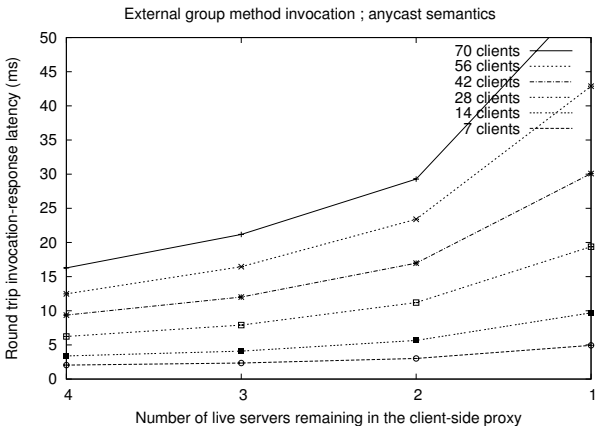

Figure 2: The performance drop due to not updating the client-side membership.

3.2 Client-side Update Delays

To be able to reason about the timing involved in updating the client-side proxy (G^A), lets assume that the proxy is updated in some unspecified manner. The timeline in Figure 3 illustrates one possible failure/recovery scenario in which the client-side proxy (G^A) is updated. In this example a group of three members install a view $V_0 = \{S_1, S_2, S_3\}$, and bind their remote references in the registry service, allowing clients to obtain a group reference (the G^A) from the registry. After some unspecified time S_2 crashes, rendering all existing G^As inconsistent with the actual situation. Given that we have two remaining servers in the group, the G^A can simply failover to another server, to perform a client invocation.

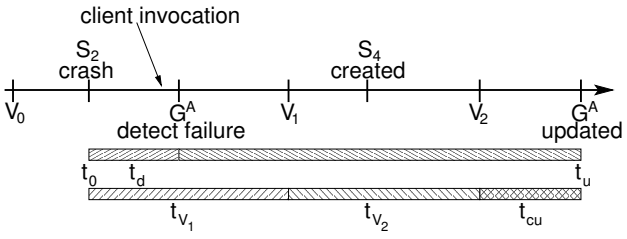

Figure 3: The update timeline

The G^A should however be updated to enable the use of all available servers, and the timeline in Figure 3 illustrates the timing involved in updating the client-side proxy. Let t_d be the time until the client-side proxy (G^A) detects that S_2 has crashed. The total time for updating G^A is given by $T_U = t_u - t_0$, and we denote this time as the *total update delay*. Note that t_d may stretch beyond t_u, when G^A does not select to use S_2, or if the client does not perform any invocations in the range $[t_0, t_u]$. Finally, let the *client update time*, t_{cu}, be the time from the installation of the compensation view (V_2) and until G^A is again consistent with the actual situation.

The failover latency, t_f is the additional time imposed on clients when attempting to invoke a server that has failed. Let t_f be the time between the proxy receiving the actual invocation and the time of performing a reinvocation on a different server, as illustrated in Figure 4. Obviously, the failover latency does not include the actual invocation-response latency.

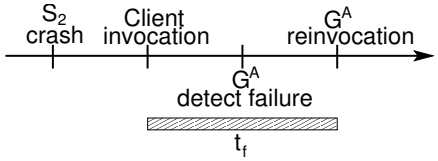

Figure 4: The failover latency

4 Updating the Dependable Registry

The current implementation of the dependable registry described in Section 2 lack support for updating its content to ensure consistency with the server-side group membership. To better understand why this is a problem, consider the following scenarios: (i) a server leaves the group voluntarily; and (ii) a server leaves the group involuntarily by crashing or partitioning. In the former case, the server may perform the unbind() method on the registry, allowing the registry database to be updated accordingly, by removing the server's reference. In the latter case, however, the failed server is unlikely to be able to perform the unbind() method. Thus, rendering the registry database in an inconsistent state with respect to the server-side membership. In this situation, the dependable registry will continue to supply clients requesting a reference (G^A) for the server group, through the lookup() method, with a proxy that contain servers that are no longer member of the group. In fact, the proxy may be completely obsolete, when all servers in the group have crashed. Furthermore, if new servers were to be started to replace failed once, the number of members of the group proxy for G^A would grow to become quite large. Figure 5 illustrates the problem visually.

To prevent clients from obtaining obsolete proxies from the dependable registry, we provide two distinct techniques for maintaining consistency of the registry content. The techniques are implemented as separate layers that must be embedded within the server group protocol stack. A detailed description of the following techniques can be found in [5].

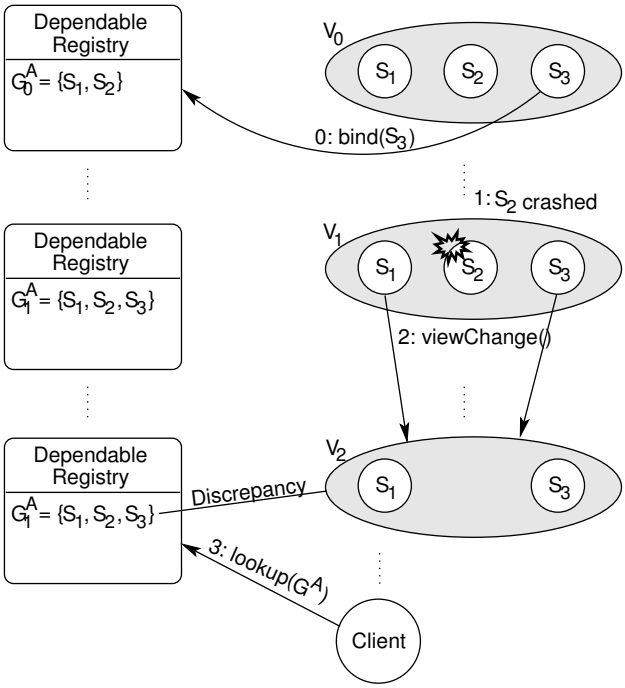

Figure 5: Example in which the registry becomes inconsistent, and a client obtains a group proxy G_1^A which is an incorrect representation of the actual group membership.

4.1 The Lease-based Refresh Technique

Our first solution to the problem is based on the well known concept of leasing. By leasing we mean that each server's object reference in the dependable registry is only valid for a given amount of time called the leaseTime. When a server's object reference has been in the registry for a longer period of time than its leaseTime, it is considered a candidate for removal from the registry database. To prevent such removal, the server has to periodically renew its lease with the registry. The interval between these refresh() invocations is referred to as the refreshRate. Figure 6 illustrate the workings of the LeaseLayer.

4.2 The Notification Technique

Jgroup provides a group membership service that allows servers (or layers) to receive notification of changes to the group's composition. These notifications come in the form of viewChange() method invocations. Thus, upon receiving such a view change event, the NotifyLayer selects a leader (S_3) for the group. The leader is responsible for updating the dependable registry in case the new view represent a contraction of the group's membership. This is done by executing the unbind() method (with S_2 as argument) on the registry.

Figure 7 illustrates the workings of the NotifyLayer.

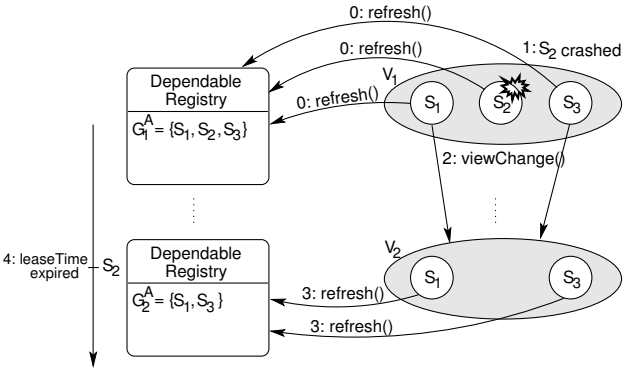

Figure 6: LeaseLayer exemplified. S_2 has crashed and is excluded by the registry since its lease is not renewed.

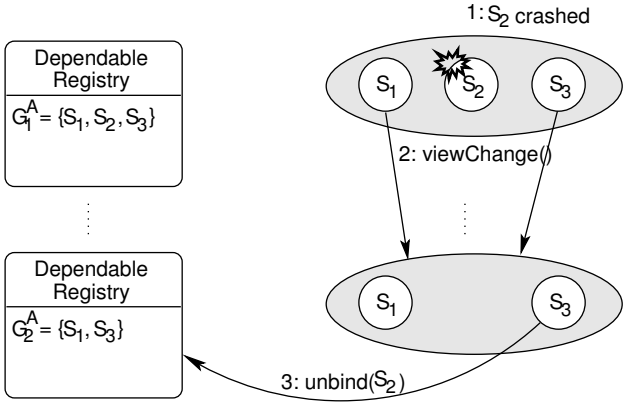

Figure 7: NotifyLayer exemplified. S_3 is the leader.

4.3 Combining Notification and Leasing

The NotifyLayer is by far the most interesting and elegant technique, but it has a drawback in situations when there is only one remaining server in the object group. In this case the last server will be unable to notify the registry when it fails. However, we may combine the LeaseLayer and the NotifyLayer, in order to exploit the advantages of both layers. The default for the combined approach would be to use the NotifyLayer (i.e., when $\#Servers > 1$), and in the situation with only one (i.e., when $\#Servers = 1$) remaining server the LeaseLayer is activated. By doing this we will also diminish the main drawback of leasing technique, namely the amount of generated network traffic, since there is only one server that needs to perform a refresh() invocation.

5 Updating the Client-side

Even though the dependable registry is kept up-to-date, the client-side proxy representation of the group membership will become (partially) invalid since it is not updated in any way. Since the membership information known to the client-side proxy may include both failed and working servers, the proxy may hide that some servers have failed by using those that work. For each *anycast* invocation, the client proxy selects uniformly a single server among the working servers.

In the current release of Jgroup, the client-side proxy mechanism will throw an exception to the client application if all group members have become unavailable, the client application can contact the registry in order to obtain a fresh copy of the group proxy. However, this yields a poor load distribution among the servers and the operation should also be transparent to the client application programmer. Various techniques could be used to obtain such failure transparency.

1. *Periodic refresh.* The client-side proxy must request a new group proxy from the dependable registry at periodic intervals.

2. *Client-side view refresh.* For each invocation, the client-side proxy attaches its current view identifier. The server compares the client view with its own view. If the two differs, the server augments the result message with its current view, allowing the client to update its membership information.

Technique 1 requires selecting a suitable refresh rate interval, which can be difficult. If set too low, it could potentially generate a lot of overhead network traffic, and if set too high we run the risk that it is not updated often enough to avoid exposing server failures to the client. This technique, work by indirect updates in that it rely on the registry already being updated, which may not be the case, depending on the technique used to update the registry. Therefore, technique 2 is very appealing in that it will work even though the registry is not updated, since the server-side handles updating clients itself.

The main difference between these techniques is the time it takes the client-side proxy to return to a consistent state with respect to the membership of the server group. This is not an issue concerning server availability, but rather the ability of the client-side proxy to load balance its invocations on all active servers, and not just the ones that are known to the clients.

6 Measurements and Evaluation

Figure 8 gives an overview of the experiment configuration, consisting of some 56 clients, each of which perform a continuous stream of *anycast* method invocations. The method used takes an array of 1000 bytes as argument and returns the same array. In each experiment we measure the round trip invocation-response latency at the clients. The time-axes correspond to the receive time of an invocation.

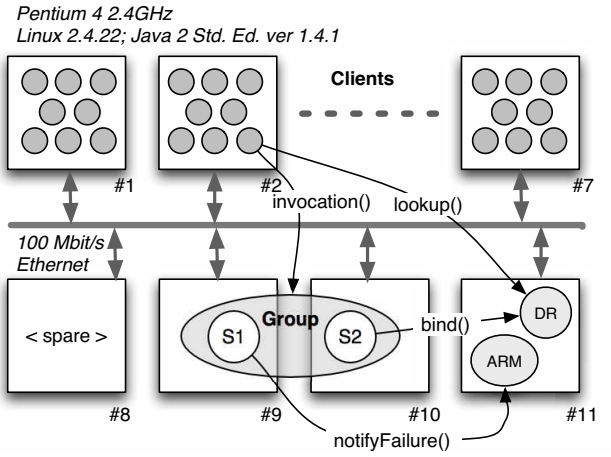

Figure 8: Experiment setup

Client-side view refresh. Figure 9 illustrate the results obtained using Technique 2. After the server crash (detected by the client at the 5 second mark), the proxy removes its reference to the failed server and continues to use only the remaining server known to the proxy. The failover latency (t_f) can be seen as the blank interval immediately before to the client failure detection point. The average $t_f = 631.8$ ms. Delayed invocations due to failover are not shown in the graph as they fall too far outside the plot range, and represents only a minor portion of all invocations. As the figure show, the actual server recovery instance, and the point at which the clients become updated is almost overlapping (difference of $t_{cu} = 52$ ms). The increase in invocation latency immediately after recovery, mainly stems from connection establishment. For readability, not all data related to post recovery is shown in the figure; that is there is also some invocation latencies in the range [200,400].

With no update, the system have a steady state performance as the one between failure and recovery in Figure 9, cf. the discussion in Section 3.

Periodic refresh. Figure 10 shows the results of Technique 1. In this experiment, we used a refresh rate of 15 seconds. As in an operative system, the clients start independently and asynchronously. Hence, re-

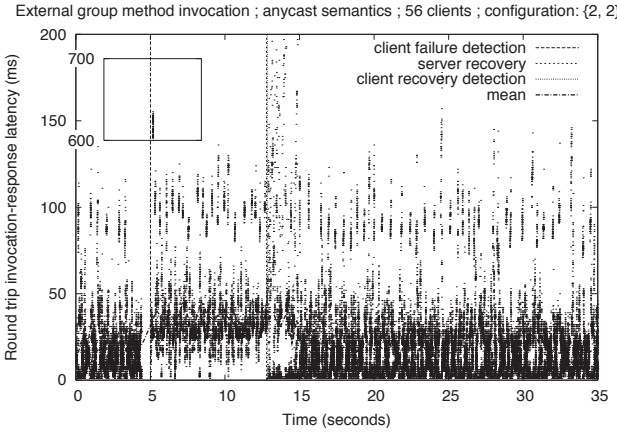

Figure 9: Client-side view refresh

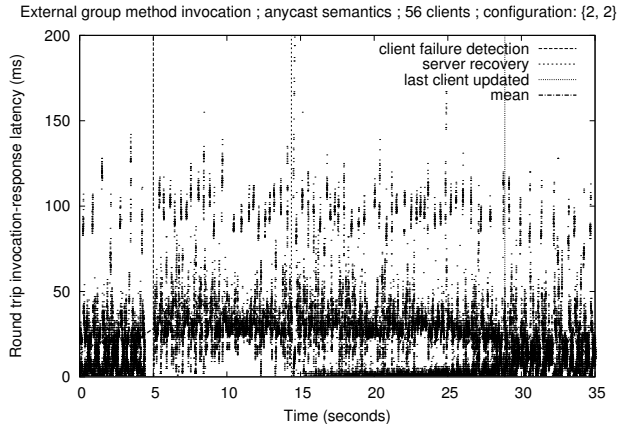

Figure 10: Periodic refresh

freshes will occur uniformly distributed over the refresh interval and the clients gradually establish connections to the replacement server. This leads to an average client update time as high as $t_{cu} = 8.16$ seconds.

In both experiments we observed higher invocation latencies (approx. 100 ms) at a regular rate of about 0.35 seconds. These invocations take longer to complete due to server-side garbage collection (GC) performed regularly by the Java virtual machine. In addition to these, we can also see several other stochastic components that are likely due to client-side GC and operating system interrupts, among other things.

7 Conclusions

In this paper we have identified limitations and a potential performance bottleneck in a client-side proxy, and its corresponding naming service implementation. This bottleneck may lead to significantly larger invocation latency for clients, and may render server failures visible to the client application, an undesirable property in middleware frameworks. Several techniques have been proposed for maintaining consistency between the group membership and its representations within the dependable registry and residing at clients. A performance study has been carried out to reveal the impact of inconsistent client-side proxies, and to compare our proposed techniques. The client-side view refresh technique, was shown to be the most effective approach.

References

[1] G. V. Chockler, I. Keidar, and R. Vitenberg. Group Communication Specifications: A Comprehensive Study. *ACM Computing Surveys*, 33(4):1–43, Dec. 2001.

[2] C. Karamanolis and J. Magee. Client-Access Protocols for Replicated Services. *IEEE Transactions on Software Engineering*, 25(1), Jan. 1999.

[3] S. Maffeis. The Object Group Design Pattern. In *Proc. of the 2nd Conf. on Object-Oriented Technologies and Systems*, Toronto, Canada, June 1996.

[4] H. Meling and B. E. Helvik. ARM: Autonomous Replication Management in Jgroup. In *Proc. of the 4th European Research Seminar on Advances in Distributed Systems*, Bertinoro, Italy, May 2001.

[5] H. Meling, J. A. Lind, and H. Hommeland. Maintaining Binding Freshness in the Jgroup Dependable Naming Service. In *Proc. of Norsk Informatikkonferanse (NIK)*, Oslo, Norway, Nov. 2003.

[6] H. Meling, A. Montresor, Ö. Babaoğlu, and B. E. Helvik. Jgroup/ARM: A Distributed Object Group Platform with Autonomous Replication Management for Dependable Computing. Technical Report UBLCS-2002-12, Dept. of Computer Science, University of Bologna, Oct. 2002.

[7] A. Montresor. A Dependable Registry Service for the Jgroup Distributed Object Model. In *Proc. of the 3rd European Research Seminar on Advances in Distributed Systems*, Madeira, Portugal, Apr. 1999.

[8] A. Montresor. *System Support for Programming Object-Oriented Dependable Applications in Partitionable Systems*. PhD thesis, Dept. of Computer Science, University of Bologna, Feb. 2000.

[9] N. Narasimhan. *Transparent Fault Tolerance for Java Remote Method Invocation*. PhD thesis, University of California, Santa Barbara, June 2001.