Maintaining Binding Freshness in the Jgroup Dependable Naming Service

Hein Meling meling@acm.org
Jo Andreas Lind jalttojo@hotmail.com
Henning Hommeland he-homm@online.no

Department of Electrical and Computer Engineering
Stavanger University College
N-4068 Stavanger, Norway

Abstract

In this paper, we discuss issues related to maintaining consistency (and freshness) between the dynamic membership of a replicated server, and its representation in a naming service storage. We propose two solutions based on leasing and notification, and evaluate the suitability of each solution based on measurements of performance impact and failover delay. Furthermore, we discuss several approaches to ensure that the dynamic membership of the replicated server is also reflected (refreshed) in the client-side representation of the server group membership.

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 accomplish these goals, it is common to *replicate* critical system components in such a manner that certain functions are performed by multiple independent replicas. Given that the number of simultaneous replica failures can be bounded, we can provide certain guarantees concerning availability and correctness in our system. Physical distribution of the replicas is an effective measure to render failures independent.

Distributed object-based middleware platforms such as CORBA [11], Java RMI [12], Jini [1] and J2EE [13] hold the promise of simplifying network application complexity and development effort. However, they remain unsuitable for implementing replication since the required "one-to-many" interaction model among objects has to be simulated through multiple one-to-one interactions. This not only increases application complexity, but also degrades performance. To overcome this limitation, distributed objects may be replaced by their natural extension *distributed object groups* [3, 5]. Clients interact with an object group transparently through remote method invocations (RMI), as if it were a single, non-replicated remote object. Global consistency of the object group is typically guaranteed through a *group communication service* [2].

Clearly all distributed middleware platforms must provide some means for clients to obtain access to a remote server object. Typically this is accomplished using a naming service, also referred to as a registry service [8]. The task of a naming service is to map a textual service name to a remote object reference, that implements a service described by

the service name. The remote object reference, also called stub or proxy, can be used by a client to access the service functionality implemented by the remote object.

Building a dependable distributed middleware platform requires also the naming service to be fault tolerant, so as to ensure that clients can always access the service. Several existing middleware platforms provide a dependable naming service, including Jgroup [8] and Aroma [10]. However, none of these dependable naming services update their database of object references in the presence of replica failures. Thus clients may become unable to communicate with the service even though there are available replicas to service client requests. Another, perhaps more important problem with current solutions, is that the client-side proxy may not have the most current membership information. The latter problem, will render failed server replicas visible to clients; an undesirable property in a fault tolerant system in which failure transparency is an important goal.

This paper presents an extension to the Jgroup/ARM [9, 7] object group middleware platform and its dependable naming service [8], for the purpose of maintaining freshness of object references stored in the naming service database. Two distinct techniques are proposed to solve this problem, one based on leasing and another based on notifications from the Jgroup membership service. In addition we have implemented a combined approach, gaining the benefits of both techniques. Furthermore, we present measurement results providing indication of the imposed processing- and failover delay of each individual technique. Finally, we propose several extensions to the client-side proxy mechanism to circumvent the problem with obsolete membership information.

The rest of this paper is structured as follows. Section 2 gives an overview of the Jgroup/ARM replication management framework, while in Section 2.1 the current Jgroup dependable registry service is presented. Section 3 presents the suggested techniques for maintaining binding freshness in the dependable registry, and in Section 4 we provide measurement results and an informal evaluation of each technique. In Section 5 we discuss the possible solutions to the client-side proxy freshness problem. Section 6 concludes the paper.

2 The Jgroup/ARM Dependable Computing Toolkit

Jgroup [9] 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. Jgroup promotes *environment awareness* by exposing network effects to applications that best know how to handle them. If they choose, operational objects continue to be active even when they are partitioned from other group members. This is in contrast to the *primary partition* approach that hides network effects as much as possible by limiting activity to a single *primary* partition while blocking objects in all other partitions. In Jgroup, applications become *partition-aware* through *views* that give consistent compositions of the object group within each partition. Application semantics dictate how objects should behave in a particular view. Jgroup also includes a *state merging service* as further support for partition-aware application development. Reconciling the replicated application state when partitions merge is typically one of the most difficult problems in developing applications to be deployed in partitionable systems. While a general solution is highly application dependent and not always possible, Jgroup simplifies this task by providing systematic support for certain stylized interactions that frequently occur in solutions. Jgroup is unique in providing a uniform object-oriented programming interface (based on RMI) to govern *all* object interactions including those

within an object group as well as interactions with external objects. Other object group systems typically provide an object-oriented interface only for interactions between object groups and external objects, while intra group interactions are based on message passing. This heterogeneity not only complicates application development, but also makes it difficult to reason about the application as a whole using a single interaction paradigm.

Most object group systems, including Jgroup, do not include mechanisms for distributing replicas to hosts or for recovering from replica failures. Yet, these mechanisms are essential for satisfying application dependability requirements such as maintaining a fixed redundancy level. ARM [6, 7] 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 provides a simple interface through which a management client can install and remove object groups within the distributed system. Once installed, an object group becomes an "autonomous" entity under the control of ARM until it is explicitly removed. 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. The distribution scheme enables deployers to configure the set of hosts on which replicas can be created. The choices included in a replication policies involve choosing the types of faults to tolerate, the styles of replication to use and the degree of redundancy to use, among other factors. The ARM implementation is based on a *correlation mechanism*, whose task is 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. The properties of our framework as described above lead to what we call "autonomous replication management".

2.1 The Dependable Registry Service

When a client wants to communicate with a Java RMI server (or an object group), it needs to obtain a reference (stub) to either the single server or the object group depending on what kind of server the client is trying to access. In the case of Java RMI, the Java runtime environment is bundled with a standard registry service called rmiregistry. This registry service is a simple repository facility that allows servers to advertise their availability, and for clients to retrieve stubs for remote objects (servers). Unfortunately, the rmiregistry is not suitable for the distributed object group model. There are several reasons for this incompatibility, as identified in [9]:

1. The rmiregistry constitutes a single point of failure.
2. It does not support binding multiple server references to a single service name, as required by a object group system.
3. Only local servers are allowed to bind their stubs in the rmiregistry.

To address these incompatibilities, Jgroup includes a dependable registry service [8, 9], that can be used by servers to advertise their ability to provide a given service identified by the 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 is designed as a replacement for the rmiregistry,

and only minor modifications are required on both client and server side to adopt the dependable registry. The dependable registry service is in essence an actively replicated database, preventing the registry from being a single point of failure. The dependable registry servers are replicated using Jgroup itself, and clients access the registry service using external group method invocations. 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:

$$ServiceName \rightarrow \{S_1, S_2, \ldots, S_n\}$$

were S denotes a server, and n represents the number of servers registered under $ServiceName$ in the dependable registry, as providing the service associated with $ServiceName$.

3 Maintaining Freshness in the Registry

The implementation of the Jgroup dependable registry as described in Section 2.1 lacks an important property with respect to dependability, namely ensuring freshness of its content. To better understand this problem, lets consider the following scenarios.

1. When a server wish to leave the server group, it will perform an unbind() invocation against the dependable registry and then exit. In this case, the registry database is updated accordingly, by removing the object reference for the server performing the unbind().

2. However, if a server leaves the server group in an abnormal manner, such as a crash or partitioning, it will be unable to perform the unbind() invocation. Thus, rendering the registry database in an incorrect state with respect to available server group members.

In the latter case, the dependable registry will continue to supply clients requesting a reference 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.

We can also describe the problem more formally. Let G_x^A represent a group proxy for group A as stored in the registry, and let the index x denote a sequence number for changes to the membership of group A as stored in the registry. Currently, only incremental changes to a group's membership will be reflected in the registry, unless a server voluntarily leaves the group. In the following we omit the group index, referring only to group A. Furthermore, let V_y denote the group membership view, and index y denotes the sequence number of views installed by the group. Using this notation, we can illustrate the second scenario above in which a server crashes. After having installed three server replicas we may have a group proxy $G_1 = \{S_1, S_2, S_3\}$ and a view $V_1 = \{S_1, S_2, S_3\}$. Now, let server S_2 fail by crashing or partitioning, leading the Jgroup membership service to install a new view $V_2 = \{S_1, S_3\}$. Since the registry will not be updated on the basis of such an event it is easy to see that the registry has become inconsistent with respect to the actual situation since $V_2 \neq G_1$, and will remain inconsistent indefinitely. 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 1 illustrates the problem visually.

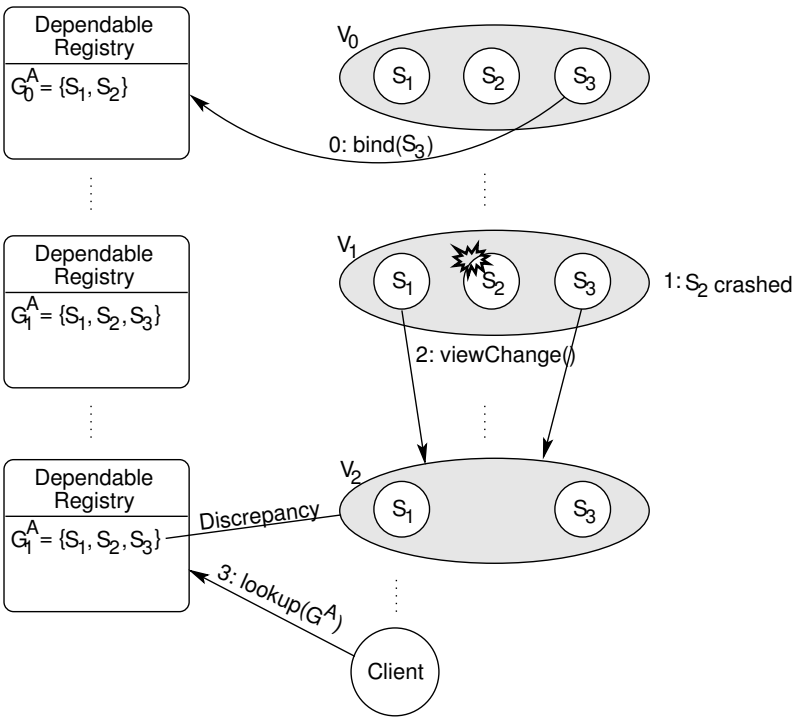

Figure 1: Exemplification of the problem, in which the registry becomes inconsistent, and a client obtains a group proxy G_1^A which is an incorrect representation of the actual group membership.

To prevent clients from obtaining obsolete proxies from the dependable registry, we provide two distinct techniques and one combined technique for ensuring freshness of the registry content. The first two techniques are implemented as separate layers that must be embedded within the server group protocol stack, while the combined approach use both layers. A detailed description of the following techniques can be found in [4].

3.1 The Lease-based Refresh Technique

Our first solution to the problem is to use 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 lease time. When a server's object reference has been in the registry for a longer period of time than its lease time, 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 refresh rate. It is possible to configure both the leaseTime and the refreshRate of the leasing mechanism, through an XML based configuration file. This is similar to the approach used in the Jini lookup service [1], except for the fact that Jini can only associate a name with a single server.

LeaseLayer Implementation

The lease-based refresh technique is implemented as a server-side layer called LeaseLayer. The layer contains a single thread that periodically performs a refresh() invocation on the dependable registry. The refresh() method is implemented in the dependable registry, and simply updates the timestamp for the current server's object

reference in the registry database. Figure 2 illustrate the workings of the LeaseLayer.

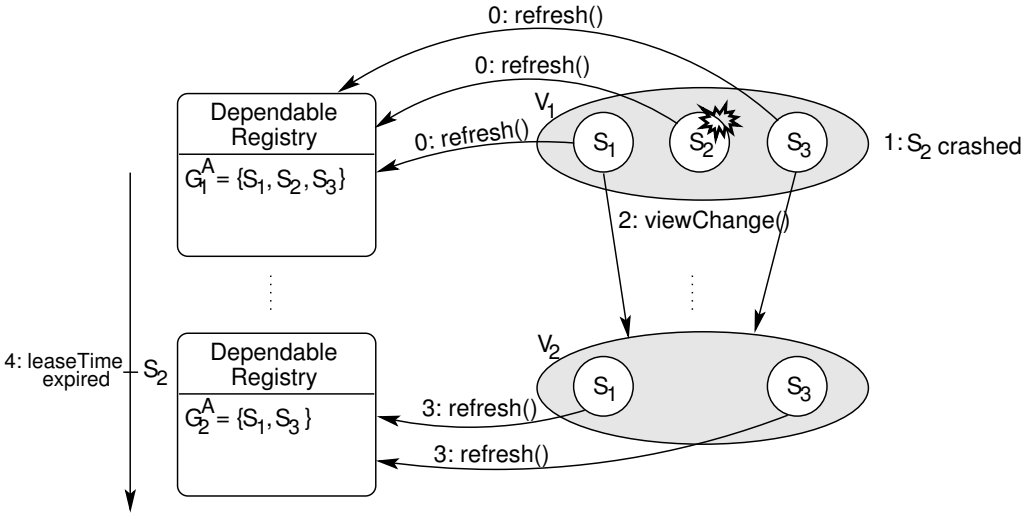

Figure 2: LeaseLayer exemplified. S_2 has crashed an is excluded by the registry since its lease is not renewed.

3.2 The Notification Technique

The Jgroup group communication system provides a group membership service, and servers that wish to receive notification of changes to the group's composition, can indicate this interest by implementing the viewChange() method of the MembershipListener. The viewChange() method is used to notify group members of changes to the group's composition, for example when a server join or leave the group. Through this view change mechanism, servers are informed of other servers in the group. This means that the servers will at all times know about the existence of other servers in its group. Thus, the objective of the notification technique is to exploit the view changes that occur within a group to inform the dependable registry about changes in the group's composition, and from that information update the registry database.

NotifyLayer Implementation

The NotifyLayer is also implemented as a server-side protocol layer. Figure 3 illustrates the workings of the NotifyLayer in a crash failure scenario. The layer intercepts viewChange() invocations from the Jgroup membership service, and a single member (the leader (S_3)) of the group invokes the unbind() method, identifying the crashed member to the dependable registry in case the new view represents a contraction of the group's membership.

3.3 Combining Notification and Leasing

The NotifyLayer is by far the most interesting and elegant technique, but it has one major flaw, namely the situation 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. Using a hybrid approach which combines the LeaseLayer and the NotifyLayer, in order to exploit the advantages of both layers. The default for the hybrid approach is 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. This hybrid approach enable the registry to serve clients with correct group proxies in most situations.

Table 1 summaries our findings for the LeaseLayer, NotifyLayer and a *hybrid approach*, and demonstrates the advantages of combining these layers. The first column (*Messages*) refers to the amount network traffic generated, while the second column (*Detection Speed*) refers to the expected time for the registry to become updated. The third column (*Single Member*) refers to the registry's ability to be consistent even after the last member of a group has crashed.

Table 1: Comparing the three refresh approaches.

	Messages	Detection Speed	Single Member
Lease	-	-	+
Notify	+	+	-
Combined	+	+	+

4 Measurements and Evaluation

In this section we present measurement results and analysis of the two refresh techniques. We are particularly interested in the failover delay and (processing) performance impact of introducing these techniques into the server-side of the group members. We define the *failover delay* to be the time between the actual failure of a server and the point at which the registry returns to a correct state. In the case of using no refresh techniques, the failover delay will be infinite; so it is obvious that any refresh technique will be an improvement with respect to the failover delay. Therefore, it is also important to consider the performance impact of the techniques used.

To conduct the various measurements presented below, we used a cluster of six P4 2.2 GHz Linux machines interconnected over a 100Mbit LAN. We ran a series of tests using a *replicated echo server* and corresponding client. The echo server exports a method that takes an array argument of varying size, and returns it. The performance

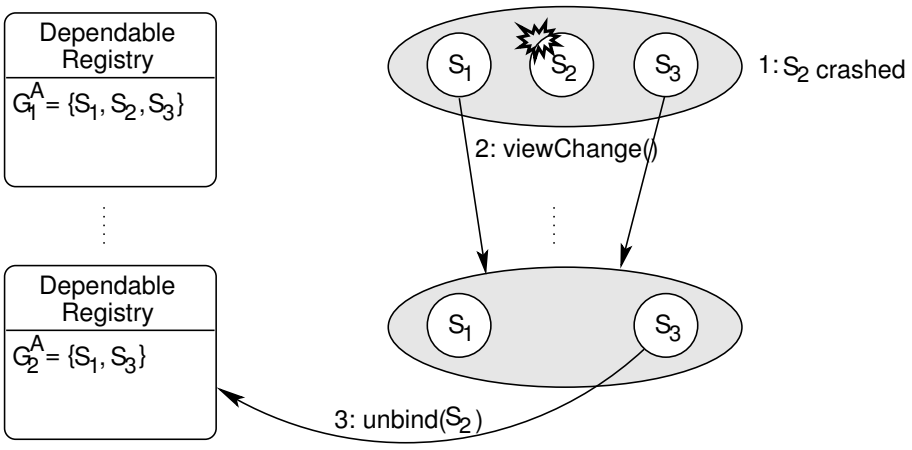

Figure 3: NotifyLayer exemplified. S_3 is the group leader.

tests involved performing both *anycast* and *multicast* invocations, and measuring the invocation delay. We obtained average, maxima and minima values for the various tests.

4.1 LeaseLayer Performance Evaluation

Evaluating the performance of the LeaseLayer, we installed a four-way replicated echo server and a single client continuously executing invocations towards the server group. A single registry instance were running on a separate machine. Each echo server was configured with the LeaseLayer, and numerous tests were run with refresh rates ranging from 1 second to 180 seconds. We want to determine if leasing have an impact on the performance of the servers, therefore we also ran the same tests with the echo server configured without the LeaseLayer. To determine if there is a performance penalty of using leasing in the given configuration, we examined the two extremes; (i) no leasing, and (ii) a refresh rate of 1 second.

We expected that such a short refresh rate would impact the performance of the server, but our comparison reveals no conclusive differences that indicates performance deterioration. The reason for this is twofold: (i) the registry is not heavily loaded, and thus can deal quickly with refresh() invocations, and thus (ii) the servers spend very little time dealing with refresh() invocations. The latter means that the servers have more than enough time to service clients, and thus we see no performance impact in our measurements. Changing the server load by adding more clients, or by using a smaller refresh rate, it is likely that there would be a difference between the two scenarios.

Note that our measurements does not explicitly attempt to measure the network traffic generated by the refresh() invocations. However, our measurements would have implicitly captured any network performance degradation, by the refresh() method taking longer to complete.

Refresh Delay

The above results made us want to examine the joint processing and network delay imposed by the refresh() method. Therefore, we setup an experiment with a single registry and a single echo server configured with the LeaseLayer and a refresh rate of 1 second. We measurement the delay of some 1000 refresh() invocations, from ten independent runs of the experiment.

The results of the experiment are shown in Table 2. As seen from the table, the average time for a server to perform a refresh() against the registry is 6.957 milliseconds. Considering that the refresh rate is as low as 1 second, the refresh processing takes up less than one percent of the overall processing time of the server. That is, the percentage of time spent performing refresh every interval is 0.691%, leaving 99.309% of the time for processing client requests.

The results obtain in this experiment strengthens our conclusions from the first experiment; the time it takes to perform a refresh() is very small relative to the total time between refresh() invocations. Note that in this experiment we used only a single echo server. If we had used more servers, a possible bottleneck could have arose; the dependable registry could have prolonged the refresh time if it had received simultaneous refresh requests from several servers. This, however, is not very likely since the probability that two or more servers will perform the refresh at the same time is very small. From the servers perspective the inclusion of leasing does not add a significant overhead, but from the perspective of the dependable registry in a system containing a multitude of servers all employing leasing, the amount of traffic directed towards the

registry can grow significantly. But even if leasing imposes an extra burden on the registry it is probable that the registry will be able to handle this extra burden. This is because the service provided by the registry is not very processor intensive (it simply serves out group proxies to clients). In conclusion, leasing in a Jgroup distributed system is feasible even with a large number of servers.

Table 2: Measurement results; refresh delay and failover delay (in milliseconds).

	Mean	StdDev	Variance	Min	Max
Refresh Delay (Lease)	6.957	0.265	0.070	6.515	7.455
Failover Delay (Notify)	86.675	87.753	7700.635	8.000	401.000

4.2 NotifyLayer Performance Evaluation

The nature of the NotifyLayer is completely different from the LeaseLayer in that it does not do anything unless there is a change in the groups membership; that is the installation of a new view. Thus, on the server-side there will not be any processing or networking overhead during normal operation of the NotifyLayer.

4.3 NotifyLayer Failover Delay

The failover delay time portion is the time it takes for the registry to discover that a server has crashed/partitioned, as illustrated in Figure 4. To determine the failover delay when using the NotifyLayer we setup an experiment with a single registry and four echo servers. We then simulated server crashes by manually stopping the server using *CTRL-C*. Once the available servers had been exhausted, the servers were brought back up again, and the tests were repeated. The failover delay was measured by the leader of the group. Note that, we were not able to measure the delay between the actual server crash and the installation of the new view (e.g., V_1 in Figure 4). However, even without this part of the failover delay, we obtained useful information concerning the delay involved. As shown in Table 2, the failover delay is on average 86.7 ms, and thus only during this short period may clients retrieve an incorrect proxy from the registry. Note that the failover delay is highly variable with a large span from maxima (401 ms) to minima (8 ms), and a standard deviation of 87.7. The reason for this variability is that the measurements were performed with varying redundancy ranging from 4 to 1 group members. Thus, we expect that the variability will increase even more when the redundancy is increased.

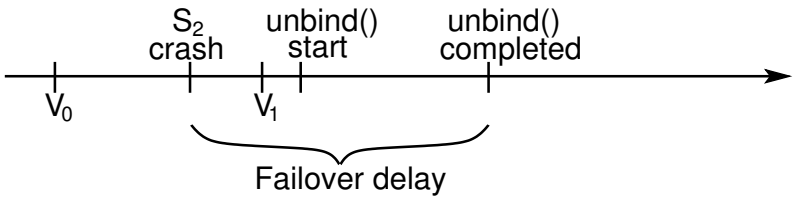

Figure 4: Illustrates the failover delay. Here we assume the initial view $V_0 = \{S_1, S_2, S_3\}$, and that after S_2 has crashed a view $V_1 = \{S_1, S_3\}$ is installed.

4.4 LeaseLayer Failover Delay

It is not easy to measure the failover delay for the LeaseLayer, since a crash may occur at the start, in the middle or at the end of a refresh period; thus the amount time between the crash and the correction of the registry table is quite variable. Therefore, we provide only a expected average for the failover delay. For the purpose of this analysis, lets assume that leaseTime = $2 \cdot$ refreshRate. This would yield an average failover delay of $\frac{1}{2} \cdot$leaseTime = refreshRate. For comparisons sake, we can easily see that the failover delay for the NotifyLayer is much smaller than the LeaseLayer, since it is dependent on the refresh rate. That is, it would not be practical to use a refresh rate as low as 100 ms, in a large scale system.

5 The Client-side Perspective

Even though the dependable registry is kept up-to-date using either the LeaseLayer or NotifyLayer (or a combination), there is still the problem of clients. When a client perform a lookup() against the dependable registry, it receives a group proxy containing a reference for each of the servers in the group. If a server crash/partition, the registry will be updated with the help of LeaseLayer or NotifyLayer. However, 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 can hide the fact that some servers have failed by using those that work. For each invocation, a single server is selected among the working servers following a uniform distribution. Once a server failure is detected, that server is marked as unavailable, and another server from the working set is selected in its place.

However, in the current release of Jgroup, the client-side proxy mechanism will throw an exception to the client application once all group members have become unavailable, rendering server failures visible to the client application. The latter is undesirable in a fault tolerant system, in which failure transparency is an important goal. The solution to this is simple enough, the client can contact the registry in order to obtain a fresh copy of the group proxy. However, this operation should also be transparent to the client application programmer. Various techniques could be used to obtain such failure transparency.

1. *No client-side proxy refresh of membership information.* This is the technique currently used in the Jgroup client-side proxy that will expose server failures to the client application. We list this technique as a comparative reference.

2. *Periodic refresh.* The client-side proxy must request a new group proxy from the dependable registry at periodic intervals. This approach requires selecting a suitable refresh rate interval. 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.

3. *Refresh when all known group members have become unavailable.* If the client-side proxy is unable to connect to a server member, it will try to connect to each of the other group members, until all members have been exhausted. That is, no more live members are available using the current membership information known to the proxy. Once this happens, obtain a new proxy from the dependable registry.

4. *Refresh when N of the known group members have become unavailable.* If the client-side proxy has tried and failed connecting to N servers, it requests a new group proxy from the dependable registry.

5. *Refresh when the client-side proxy detects a new server-side view.* For each invocation performed by a client, the server-side proxy attach its current view identifier (viewId) together with the result of the invocation to the client-side proxy. In this way, the client-side proxy is able to determine if it needs to contact the dependable registry to obtain fresh group membership information.

Technique 2 is not really interesting since it is difficult to determine a suitable refresh rate, and it is unlikely to scale well for a large number of clients. Technique 3 is a special case of Technique 4, in which $N = all$. The main difference between the last three 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. In all of the above techniques, there will be a slight failover delay once a client-side proxy detects (during an invocation) that a server has failed. However, in Technique 5 the time to detect a new view is typically faster than the other techniques, since it will detect it as soon as it invokes a working member of the group after a new view has been installed. The latter approach may be combined with Technique 4 (e.g., for $N = 1$), to update the proxy also when a failed server is detected before a new view.

Technique 5 in combination with Technique 4 is perhaps the most interesting, however currently we have only implemented technique 3.

6 Conclusions

In this paper we have identified important limitations with existing dependable naming service and client-side proxy implementations. In many server failure scenarios, these limitations will exposed failures to the client application, an undesirable property in most middleware frameworks.

We proposed two distinct techniques for maintaining freshness in a dependable naming service and provide measurement results. Our analysis of the results indicate that the performance impact and failover delay of the notification based approach is the most efficient. When used in combination with the lease based approach we can also recover from failure of the last member, and the additional performance impact of the lease based approach will be limited only to a small period when the group consists of only a single member.

Furthermore, we propose several techniques for ensuring client-side proxy freshness. Currently, we have implemented only Technique 3, and this solves the client-side proxy freshness issue. However, we conclude that a combination of Technique 4 and Technique 5 is perhaps worth investigating, since it will have a shorter failover delay and will always recover to a consistent state as long as there are more replicas available from the client location.

References

[1] K. Arnold, B. O'Sullivan, R. Scheifler, J. Waldo, and A. Wollrath. *The Jini Specification*. Addison-Wesley, 1999.

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

[3] G. Collson, J. Smalley, and G. Blair. The Design and Implementation of a Group Invocation Facility in ANSA. Technical Report MPG-92-34, Distributed Multimedia Research Group, Department of Computing, Lancaster University, Lancaster, UK, 1992.

[4] H. Hommeland and J. A. S. Lind. Maintaining Binding Freshness in a Dependable Naming Service (in the Prescense of failures). Technical report, Department of Electrical and Computer Engineering, Stavanger University College, Apr. 2003. Advisor: Hein Meling.

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

[6] 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.

[7] 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.

[8] 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.

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

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

[11] OMG. *The Common Object Request Broker: Architecture and Specification, Rev. 2.3*. Object Management Group, Framingham, MA, June 1999.

[12] Sun Microsystems, Mountain View, CA. *Java Remote Method Invocation Specification, Rev. 1.7*, Dec. 1999.

[13] Sun Microsystems, Mountain View, CA. *Enterprise JavaBeans Specification, Version 2.0*, Aug. 2001.