The following paper was originally published in the
Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)
Santa Fe, New Mexico, April 27-30, 1998

Filterfresh: Hot Replication of Java RMI Server Objects

Arash Baratloo, *New York University;*
P. Emerald Chung, Yennun Huang, Sampath Rangarajan, and Shalini Yajnik,
Lucent Technologies, Bell Laboratories

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org/

Filterfresh: Hot Replication of Java RMI Server Objects

Arash Baratloo*

P. Emerald Chung
Yennun Huang
Sampath Rangarajan
Shalini Yajnik†

Department of Computer Science
New York University, 251 Mercer Street
New York, NY 10012

Lucent Technologies Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974

Abstract

This paper presents the design and implementation of a Java package called *Filterfresh* for building replicated fault-tolerant servers. Maintaining the correctness and integrity of replicated servers is supported by a `GroupManager` object instantiated with each replica to form a logical group. The Group Managers use a *Group Membership* algorithm to maintain a consistent group view and a *Reliable Multicast* mechanism to communicate with other Group Managers. We then demonstrate how *Filterfresh* can be integrated into the Java RMI facilities. First we use the `GroupManager` class to construct a fault-tolerant RMI registry called *FT Registry*—a group of replicated RMI registry servers. Second, we describe our implementation of the *FT Unicast*—a client-side mechanism that tolerates and masks server failures below the stub layer, transparent to the client. We also present initial performance results, and discuss how general purpose RMI servers can be made highly available using the *Filterfresh* package.

1 Introduction

Distributed object technologies have become popular in developing distributed applications. Among these, object technologies such as CORBA [17], DCOM [4], and Java Remote Method Invocation (RMI) [22, 20] are the most popular. Although these middleware platforms ease the development of distributed applications, they do not directly improve the reliability of these applications. As a result, application developers have to implement their own mechanisms to improve the reliability and availability of their applications. The task of developing fault tolerance techniques for distributed object paradigms is often tedious and error-prone. Therefore, there is a great need to develop a generic, portable and reusable tool that enhances the reliability and availability of distributed objects.

In this paper we focus on using *Java RMI* to implement reliable objects. In Java RMI, an application consists of client and server objects. A client invokes a server's method using the server's object reference. To make its object reference available to clients, a server registers a tuple containing its object reference and a string name with a name-server called *RMI registry*. This operation is called *binding*. Given a string name, clients can get the remote reference of a server registered under that name by contacting the RMI registry. This operation is called the *lookup*. There could be many registries running in a network, but, registry data sets among different registries are not shared or replicated. Therefore, a client must have *a priori* knowledge of hosts running RMI registries. From the fault tolerance point of view, the current registry implementation is a single point of failure for RMI applications. For example, if one registry fails all of its data is lost and clients cannot get object references of servers running at

*baratloo@cs.nyu.edu. This work was done while the author was a summer intern at Bell Laboratories.

†emerald,yen,sampath,shalini@research.bell-labs.com.

that site anymore. As a result, even though the servers are still alive they may not be accessible. This problem becomes even more complicated when we make servers migratable which forces them to re-register after a migration. Therefore, to make it possible for clients to find servers running on remote hosts unknown to them, and to make RMI applications fault tolerant, it is necessary to enhance the registry mechanism.

One way of improving the registry mechanism is to replicate its data sets among all nodes. Clients can then query any node on a network to get a server's object reference without the need of identifying the right registry first. To achieve this, we adopted the hot replication scheme, i.e., updates to any node are reliably propagated to all other nodes, and changes are made *consistently* on all sites. This approach ensures that the registry data sets are strongly synchronized.

We use the virtual synchrony model [3] to implement *FT Registry*, our group of replicated RMI registry servers. Virtual synchrony and its underlying process group operations are provided by toolkits such as Isis [3] and Transis [6], in Java middle-ware systems such as iBus [15], and in operating systems such as Amoeba [11] for building fault-tolerant applications. Based on the success of such systems, same mechanisms are used in Orbix+ISIS [9] and Electra [13] for adding fault-tolerance to CORBA, in the work proposed by [1] for adding fault-tolerance to other Object-Oriented systems, and in systems such as [12, 14] in providing fault-tolerant distributed Name Servers. Our challenge here is to integrate such mechanisms into the Java RMI system with minimal changes while staying 100% Pure Java.

Our *FT Registry*, in addition to being able to tolerate failures itself, provides the building block for fault-tolerant RMI application servers. For example, a crashed server can be restarted on a different node and it can register with another *FT Registry*. Since the server's reincarnation (i.e., the new registration) is propagated to all nodes, any client on the network can lookup the server's new object reference. This does not solve all the problems however. In the mean time, clients holding the old object reference may invoke remote operations which will fail. To recover from such a failure, we provide *FT Unicast*. The *FT Unicast* object works below the stub layer and gets a valid object reference and retries the invocation whenever a server failure is detected, thus making the server migration and fail-over transparent to client applications.

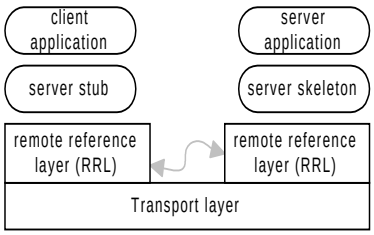

Figure 1: RMI architecture

In the next Section, we describe the Java RMI architecture. Section 3 provides an overview of the *FT Registry* and *FT Unicast* fault-tolerance mechanisms. Section 4 describes the `GroupManager` class that is used to manage the group of replicated *RMI registries*. Sections 5 and 6 describe implementation details of the two fault-tolerance mechanisms and give initial performance results. Section 7 describes how the `GroupManager` can be used to provide fault-tolerance to general Java application servers through replication. Conclusions are presented in Section 8.

2 RMI Architecture

We briefly describe the Java RMI architecture in SUN's JDK1.1 reference implementations. In a nutshell, *Java RMI* enables an object (client) to invoke methods of certain interfaces implemented by another object (server) running on a different *Java Virtual Machine* either on the same host or on a different host.

The RMI architecture consists of three layers as shown in Figure 1: the *stub/skeleton* layer, the *remote reference* layer (RRL) and the *transport* layer [22]. On the server side,

for an interface to be invoked remotely, it has to be derived from the `Remote` class. The object that implements this interface may derive from the `UnicastRemoteObject` class of the RMI package. The current `UnicastRemoteObject` uses *TCP* for low-level transport.

The *rmic* complier takes a server object implementation and generates two class definitions, a stub object for the client and a skeleton object for the server.

On the server side, when the server object is created, the constructor of `UnicastRemoteObject` performs an `exportObject()`. Inside `exportObject()`, an `UnicastServerRef` object is instantiated and exported. It creates a live reference object (the transport layer) which contains an *IP address*, a *TCP port* number and an *Object ID*. It also creates the skeleton and the stub at the server side. Then a mapping from the Object ID to the stub and skeleton is registered in an *object table* residing in the transport layer.

On the client side, the application obtains a reference to the server object from RMI registry or from other objects. If the client does not have the stub code in the local host, the stub is dynamically loaded from the server side. The stub is a layer between the application and the lower layers of the RMI mechanism. The main function of the stub is the marshaling/unmarshaling of requests and results and passing them between the client and the Remote Reference Layer. The client stub contains a `RemoteRef` object. The `RemoteRef` object encapsulates the transport layer underneath. The transport layer gets the live reference of the server object and establishes the connection to the server side. The client stub calls `invoke()` method in `RemoteRef` to make the call to the remote site. Once the call gets to the server side endpoint, the server side transport checks the object table and maps the Object ID to the corresponding skeleton to dispatch the request. The skeleton unmarshals the parameters from the request and then makes the up-call to the object. The results are marshaled by the skeleton and passed back to the client side.

RMI registry is a simple name server provided by the RMI package. A server object registers a name using the `bind()` method call. The registry keeps a name to remote object mapping. It listens at a well-known port, typically, `1099`. Any client can get a reference of a remote object by name via the `lookup()` method call.

3 Overview of *FT Registry* and *FT Unicast* Fault-Tolerance Mechanisms

Filterfresh is a Java package for building highly-available servers in presence of processes crashes and network failures. In applying *Filterfresh* to Java RMI, we have implemented a Fault-Tolerant Registry (*FT Registry*) service. This service is then used to mask server failures in RMI client/server applications at the client side, completely transparent to the client (*FT Unicast*).

3.1 Replicated RMI Registry - *FT Registry*

RMI registry with the "local registry" requirement where application servers can `bind` services only with the registry local to the server machine, is too restrictive for failure recovery. This also restricts the dynamic migration of servers from one machine to another since there is no standard method for clients to find the location of application servers. We can eliminate the problem of the registry being a single point of failure and the problem of locating application servers by replicating the registry and distributing the replicas over different machines on the network. Thus, we provide a replicated RMI registry on a network, and manage the replicas to maintain consistent data sets. We also perform failure detection of the RMI registry replicas and if the registry replicas are manually restarted, enable them to transfer state from one of the available replicas and synchronize their state.

The main problem then is to keep all replicas of the registry servers synchronized in

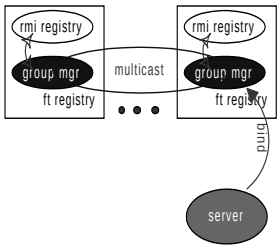

Figure 2: Server binds with the *FT Registry*

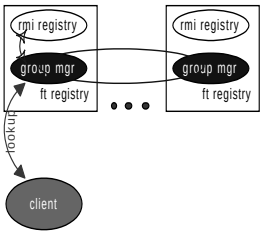

Figure 3: Client looks-up the *FT Registry*

spite of process failures and network failures. It is well known that the *process group approach* tolerates these failures. A solution based on the process group approach would provide the following.

1. Allow the replicas of the registry server to form a group.

2. Let each of the replicas maintain a consistent view of the group; i.e, let them be aware of who is in the group and who is not, in a consistent way.

3. Let the replicas propagate updates through a group multicast primitive (this is performed through a `GroupManager` class explained below); for example, if a server object `binds` with one of the registry replicas, this will be reliably propagated to other replicas so that they can update their data set to reflect this event.

4. Provide for total order on the messages that are used to propagate updates so that data sets are updated, by all replicas, in the same global sequence and hence in a consistent way; for example, if a server A *binds* with one of the registry replicas while another registry B *joins* the group of registry objects, we guarantee that the two events will be observed in the same order by all replicas. This ensures that either (1) the data set transferred to B is the image before A's registration followed by the registration event, or (2) the data set transferred to B is the image after A's registration.

The server group is managed by implementing a *Group Membership* algorithm. We provide a Java `GroupManager` class that implements the group membership algorithm using a *Reliable Multicast* primitive. The `GroupManager` object is instantiated in each replica of the *RMI registry* server as shown in Figure 2. The group managers used by the different replicas of the RMI registry form a process group. The `GroupManager` object supports the following operations.

1. *Group creation:* When a registry server is instantiated for the first time, its group manager creates a group with this as the only member.

2. *Join:* When another server replica is instantiated, its group manager *joins* the group by first transferring state from an existing replica, and then updating the group view (before any other operation can take place). This ensures uniform view and consistent states among replicated registries.

3. *Leave:* A server replica is allowed to *leave* the group.

4. *Failure detection:* The group managers *ping* other group managers periodically and if they detect a failure perform a change of view for the group.

5. *Reliable multicast:* Guarantee that messages directed to all group members are atomic and totally ordered across all replicas. The group managers themselves multicast the above group operations (such as *join*), and *FT Registry* servers multicast registry operations (such as `bind`) using the reliable multicast provided by the `GroupManager` class.

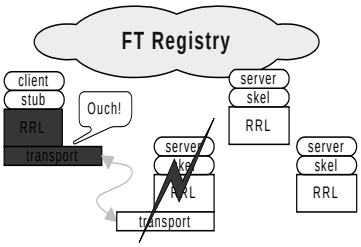

Figure 4: Application server failure detected on a remote method call

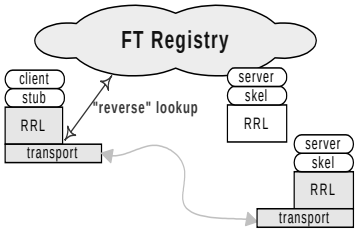

Figure 5: Stub does a reverse lookup on the *FT Registry*

Figures 2 and 3 show examples of how the `GroupManager` object is used by the *FT registry*. We adopt the *write-all-read-one* semantics. In Figure 2 when an application server binds with the local RMI registry, this information is updated locally as well as reliably multicast to all other group managers so that they can update their RMI Registry replicas. Figure 3 shows a client performing a `lookup` operation. In this case, the lookup is performed locally and is not multicast to the other group managers.

The `GroupManager` implementation is described in Section 4 and the *FT Registry* implementation using the `GroupManager` is described in Section 5.

3.2 Transparent Client-Side Fault-Tolerance - *FT Unicast*

FT Registry allows multiple application servers providing the same service and running on different hosts to register under the same name. Thus, RMI clients observe the same interface using our fault-tolerant services as with standard RMI servers, however, we use this feature to mask server failures completely transparent to clients.

We accomplish masking of server failures as follows. In a standard client/server RMI applications, a client first gets a remote reference of a server object, typically by a name lookup from the registry server. When the client makes a method call on a non-faulty application server, the stub uses this remote reference to contact the server. On the other hand if the application server has failed, an exception is raised. Now consider a set of replicated application servers registered with a group of *FT Registry* servers under the same name. If the application server has failed, the raised exception is caught at the remote reference layer as shown in Figure 4. The remote reference layer performs a `reverse lookup` at any of the registries using the *stale* reference to the faulty application server, as shown in Figure 5. The *FT registry* returns the *name* of the faulty application server. This name is used to make a normal `lookup` to get a fresh reference to an available replica of the server object. The method invocation is retried with the new server and the results are returned to the client. This provides an illusion of a valid object reference to the client. The client is unaware of the actions that the remote reference layer takes between the time it makes a remote method invocation to the time it receives the results.

The *FT Unicast* implementation is explained in more detail in Section 6. In the next section we describe the `GroupManager` class implementation.

4 Implementation of the `GroupManager`

The process group approach is at the heart of our system in providing fault tolerant services. The process group functionality is provided by the java `GroupManager`

class. To achieve fault-tolerance, any object, such as the RMI Registry, can instantiate a `GroupManager` object and use its services. In the rest of the section, we will refer to objects that instantiate and use the services of the `GroupManager` as *clients*. The `GroupManager` class ensures *atomic* and *totally ordered* group operations in presence of *crash* failures. The atomicity assures that that an event is either seen by all group members or none. The total ordering assures that all group members observe the events in the same relative order.

The protocol assumes an *unreliable* point-to-point message delivery. The communication is implemented with *UDP* [19]. *UDP* datagrams are unreliable, and hence, appropriate mechanisms such as acknowledgement, retries, and timeouts are provided at a higher level to ensure correct group operations. We chose UDP as opposed to other protocols, such as TCP, for three reasons. First, a connection-less protocol is less rigid and can tolerate transient network outages. Second, since our system had to incorporate appropriate high-level mechanisms for communication and processes failures, any buffering and retransmission by the communication layer would have been redundant. And finally because UDP is faster. In retrospect, and as the experiments will show, the performance gained by using UDP did not have a large impact on the overall system performance.

A `GroupManager` object runs its own thread of control. Client-to-`GroupManager` interactions such as multicast, are done through method invocations. On the other hand, `GroupManager`-to-client interactions, such as `GroupManager` informing a client application of receipt of a multicast message, are done through asynchronous call-back functions. Through our initial experiments with building fault-tolerant systems such as the *FT Registry*, we have found that call-backs work well in integrating group membership services into object oriented systems. We are considering other models for future implementation, in particular, the new event model introduced in Java version 1.1.

4.1 Group Operations

The `GroupManager` class implements the following five basic operations: group creation, join, leave, reliable multicast, and reset group view. We describe each operation in turn.

Group Creation: A `GroupManager` object can create a new group at any time by invoking the public method `createNewGroup()`. This invocation results in creation of a new group having the `GroupManager` object as its only member. Once it has become a group member, the `GroupManager` object can be queried for other group members, the leader of the group (described below), and it can multicast messages to all members.

Join: A `GroupManager` object that does not already belong to a group can join an existing group. The public method `joinExistingGroup()` takes a host name and a port number (of any one of the group members) as parameters. Once `joinExistingGroup()` is called, the control is passed to the `GroupManager` object and the calling thread blocks. The `GroupManager` provides the atomicity and the total ordering of the join operation by using the group *reliable multicast* operation (as described below). Once the original group members receive the join event, the state of one of the original members is transferred to the joining member. In our implementation we have found that object serialization is a convenient mechanism to implement state-transfer. After the state of the new member is brought up to date, the calling thread is unblocked.

Leave: The leave operation is implemented by the public method `leave()`. Its implementation is analogous to the join operation in blocking the calling thread, multicasting the leave event, and unblocking the calling thread when the multicast succeeds.

Reliable Multicast: In every process group there is a distinguished member called

the *group leader*. The *group leader* runs the same code as other members, and interacts with the client application the same way, the only difference is that it has more responsibility. If the group leader crashes, or if another member suspects it of crashing, the group can elect any other member to function as the new leader.

When a client application invokes the `createNewGroup()` method of an `GroupManager` object, it results in the creation of a new group. The `GroupManager` object is the only member of this group, and by default, it becomes the group leader.

A `GroupManager` object exports a public method called `multicast()` that can be invoked to send an atomic and totally ordered multicast message to all the group members. When a client invokes the `multicast()`, the message is passed to the `GroupManager` thread and the calling thread blocks. The `GroupManager` stores a copy of the message in a local buffer, then forwards it to the group leader and waits for an acknowledgement of the operation's success before unblocking the calling thread. This message is not guaranteed to reach the group leader since UDP datagrams are unreliable. For this purpose, the `GroupManager` sets up a timer and resends the message to the group leader if the timer expires before receiving the acknowledgement. When the group leader receives the message, it increments a message sequence number and sends the message along with the sequence number to all group members. The sequence number serves to ensure duplicate messages are handled properly. Once the group leader receives the acknowledgements, it notifies the object that initiated the multicast that the operation has succeeded. On the other hand, if the group leader fails to receive the acknowledgements after a set number of retries and within a given timeout period, it initiates a *reset group view* operation to recover from potential failures.[1]

[1] In practice, we observed that system performance is very sensitive to the timeout period and the number of retries. If the numbers are set too high, the system takes a long time to detect failures or to resend dropped message. For numbers that are set too low, it causes the system to send excessive messages and to initiate failure recovery too often.

From the above discussion, we see that every group operation is issued from the same process, namely the group leader, and operations are carried out one at a time. Therefore, in the absent of process crashes, every group operation is atomic and group members observe the events in the same order.

The multicast protocol that we implemented can be categorized as *ack-based* since messages require explicit acknowledgement. See [2, 10, 16, 11] for other protocols that are not *ack-based* but provide the same semantics.

Reset Group View: Informally, a *group view* refers to the list of group members that a `GroupManager` object knows about, along with the *unique id* of each member, the identity of the group leader, and a *view incarnation* number. The view incarnation number is a counter that is incremented with each view change. The view incarnation number is included in every message and it serves to ensure that a message directed to an old group will not be accepted by a new group.

Reset group view refers to a member initiating *failure detection* and wanting to reestablish the group view. This operation is generally used to recover from failures, that is, after one `GroupManager` suspects another of failure. However, a client application can, at any time, initiate a reset group view operation by invoking the public method `resetView()`. Once a `GroupManager` object enters a reset view mode it blocks all other operations until a new view is installed.

Our reset view protocol is based on [11]. It runs in two phases. The first phase of the protocol determines a new group view, i.e. establishes which members are non-faulty and chooses the group leader; the second phase of the protocol brings the members up-to-date, and then installs the view determined in the first phase.

In the first phase, any `GroupManager` that invokes the `resetView()` method becomes a *coordinator*. Thus, there may be more than one coordinator at a given time running the first phase. A coordinator invites other

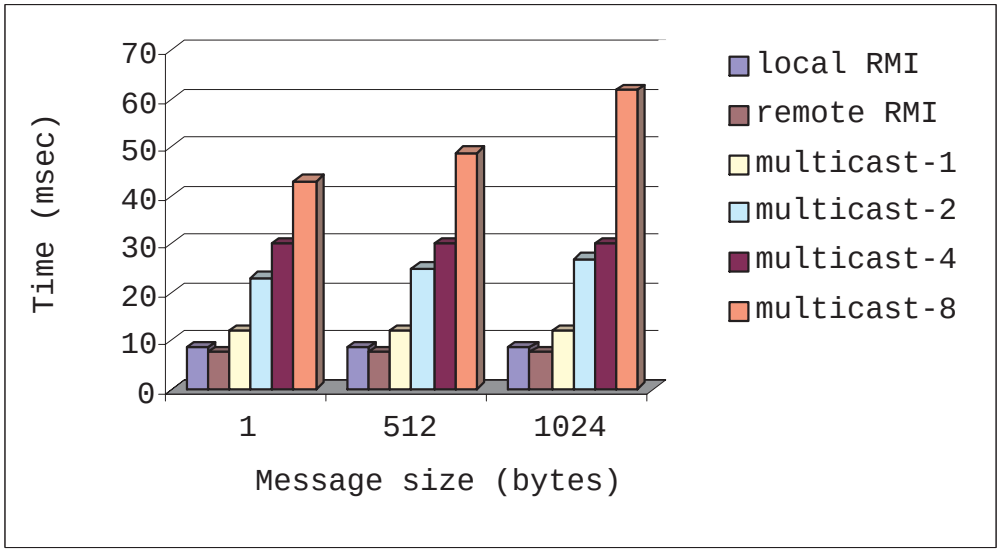

Figure 6: Performance of group multicast operation. The x-axis denotes the message size and the communication method. The y-axis represents time in milliseconds.

members to create a new view by sending a `request_view_change`. A non-faulty member that is not a coordinator accepts the invitation by responding with `ok_view_change` message. A coordinator accepts the invitation of another coordinator only if the inviting coordinator has a larger *id* number. Once a coordinator has received `ok_view_change` messages from a majority of group members, it continues to the second phase. If a coordinator is not able to successfully invite enough members within a timeout period, it repeats the first phase again. If a non-coordinator has not installed a new view within a timeout period, it becomes a coordinator and starts the first phase. Because it is required for a coordinator to successfully invite a majority of old group members, at most one coordinator could reach the second phase.

In the second phase, the coordinator first makes sure that every member has the latest message, i.e., the message with the largest sequence number. It then creates a view with the new members, the new group leader (itself), and the new incarnation number. The view is then sent to every member to install. The second phase completes when the coordinator receives an acknowledgement from every new member. If the coordinator does not receive all acknowledgements within a specified time, it repeats the first phase again.

Notice that the process of constructing a new group view will block until enough surviving group members can be found. It is well known that it is impossible to have a deterministic, correct and terminating algorithm to achieve consensus [7] in the presence of even a single failure and to build reliable failure detectors [5]. In the presence of these negative results, this protocol guarantees correctness *if and when* it terminates—that is, it will block until a consistent state can be constructed. Specifically, this protocol guarantees that if it terminates (1) all surviving members have a consistent group view, and (2) all the members in the new group view successfully receive all the messages sent by any member of the original group view before the failure.

4.2 Experiments

Here we present initial performance results for our reliable group multicast implementation. Experiments were conducted using up to 8 PentiumPro/200 machines connected by a Fast Ethernet hub. We used JDK1.1.1 running on Linux RedHat 4.0, and compiled with

optimization turned on. Reported times are *elapsed times*, and hence account for all overheads.

We measured the elapsed time for a group multicast operation to complete as measured from the invocation of the multicast until the invocation thread unblocked. This includes the time for the client to forward the message to the group leader, the group leader to reliably multicast the message, and then for the group leader to acknowledge the success of the multicast to the initiating client. We timed the operation for groups of size 1, 2, 4 and 8, and messages of size 1, 512 and 1024 bytes. With the exception of the group of size one, the multicast was initiated from an arbitrary member other than the group leader. We also measured the time for sending equivalent size messages using a single Java RMI. The results are shown in Figure 6.

Our first observation is a counter-intuitive one. We found that RMI is faster across two remote machine than on single host. We contribute this to the inefficiency of the Java runtime system we used—the faster intra-machine communication could not compensate for the shortage of resources. We were also surprised by the inefficiency of multicasts to groups of size one. When a group consists of only one object, there are message processing times, but there are no message transmissions. Multicasts took approximately 12 milliseconds. We attribute some of this to inefficient Java threads implementation under Linux. For example, we found that threads blocked on user inputs are never preempted—the work around seemed to have been expensive. Furthermore, we used object serialization for constructing low-level control messages, and as reported in [8], there is a high overhead associated with object serialization due to inefficient buffering and copying of the data. Considering that we sacrificed efficiency for simplicity in choosing the multicast algorithm, the `GroupManager` class shows reasonable scalability. For example, in increasing the group size from 1 to 8, we observed an average slowdown of 5.

5 Implementation of *FT Registry*

The `GroupManager` described in the last section is used to build the *FT Registry*. In Java RMI terminology, a *registry* is a remote object that provides a basic name server functionality. The `Registry` interface and the `LocateRegistry` classes provide this functionality. Two methods provided by the RMI registry are of special interest to us: `bind()` — to map a remote (server) object to a string (service name), and `lookup()` — to get a remote object associated with a string. The `rmiregistry` provided in JDK1.1 is a shell-script command that invokes `RegistryImpl`, an implementation of the `Registry` interface.

5.1 Replication Approach

A limiting factor of the existing RMI system is that the `RegistryImpl` successfully binds an object only if it is local to its machine. This introduces two problems in building client/server systems based on *Java RMI*. First, a client must have *a priori* knowledge of the host running the registry and the server. Second, the RMI registry becomes a single point of failure.

We address both problems by providing a replicated registry service, and by maintaining a consistent state among all replicas through the state machine approach [18]. Our implementation consists of the `FTRegistry` interface, and `LocateFTRegistry` and `FTRegistryImpl` classes. We also extends the standard interface by introducing the `multiBind()` method. The `multiBind()` method is a mechanism for multiple replicas to register under the same name. When this happens, the `lookup()` method returns an arbitrary object at random. This means that by replicating critical services, their loads will also be dispersed without client awareness, and without any effort on the part of the programmer.

Our `FTRegistryImpl` class is a replicated implementation of `FTRegistry`, replicated in the sense that instances of this class form and maintain a logical group for the du-

ration of their existence. By default, the first `FTRegistryImpl` object forms a singleton process group. Other replicas perform a group-join and a state-transfer, in which the state of the new member is brought up-to date, before becoming functional. The management and the communication among the group members are provided by embedded `GroupManager` objects.

A `FTRegistryImpl` object is composed of two logical layers: the *RMI registry* and *group manager* layers. The registry layer contains the actual data structures for object name-reference mappings. It is through private methods implemented at this layer that mappings can be added, removed and queried. The public methods such as `lookup()`, `bind()` and `multiBind()` are wrappers. When such methods are invoked, depending on whether the operation *alters the state of the registry map*, they are either passed up to the registry level to execute the corresponding private methods on the local data, or passed down to the group manager layers to multicast to all replicas.

To ensure consistent states across all `FTRegistryImpl` objects, operations that alter the registry map must be executed by all replicas. For illustration purposes, consider the case when a server object invokes the `bind()` operation of a `FTRegistryImpl` running on its local host. This is depicted in Figure 2. The group manager inspects the method and determines that the execution will result in modification to the registry map. Since the operation needs to be executed by every replica, the group manager sends the event to others using the `GroupManager`'s group multicast. This ensures that the maps of all registry replicas contain the same information at all times.

The hot replication of registry maps has two clear advantages. First, the RMI naming service will no longer remain a single point of failure. Second, it simplifies the `lookup()` operation. Clients no longer need *a priori* knowledge of the server's host, since a `lookup()` operation performed by any registry (see Figure 3) will return a server registered anywhere on the network.

5.2 Supporting `Reverse Lookup`

As mentioned earlier, we have implemented a system that can transparently mask server failures. Because of the *transparency* requirement, we had to work below the code that is generated by the Java and RMI compilers, see Figure 1. For this purpose, a fault has to be detected and masked at the `Remote Reference Layer` (*RRL*) or below. However, at the *RRL* level, concepts of server objects and server names do not exist, there are only remote reference objects and connections. Thus we need a mechanism that could construct a connection to a replicated server, given a *stale* connection to a crashed server.

We addressed this problem as follows. First we extend the mappings of our *FT Registry*. For each server object, in addition to storing its name and remote object, we store its connection object (live reference). With the added information a *reverse lookup* operation, where a server name can be looked up based on its live reference, becomes possible. Thus, given a server's live reference, our *FT Registry* can return references to other replicas of that server. In the next section we will discuss how this functionality is used.

Also note that registering the live reference is transparent to users—the server object simply calls `bind()` or `multiBind()` methods of our `DistributedNaming` class that extends the standard `Naming` class. Accessing the live reference and passing it to the `FTRegistryImpl` are hidden in our implementation.

5.3 Experiments

We measured the time for `bind()` and `lookup()` operations using the RMI registry provided with JDK1.1, and using our *FT Registry*. Experiments were conducted in the same setting as in Section 4.2. The results are shown in Figure 7.

Our implementation of `lookup()` is fast, even when it provides a richer functionality. On the other hand, our `bind()` operation is

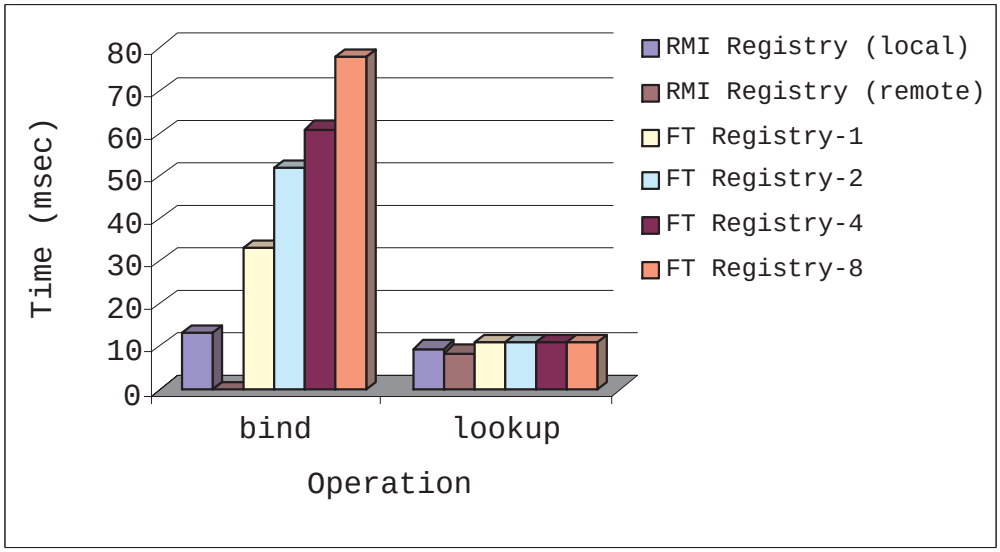

Figure 7: Performance of *FT Registry*. The x-axis contains the `lookup()` and `bind()` operations for both the standard RMI registry and our fault tolerant implementation. The y-axis represents time in milliseconds.

slower, since such events must be sent to all replicas and the new entry becomes visible at remote hosts. This functionality is not provided by the standard RMI registry service. Again, our implementation of registry service seems to scale reasonably well. For example, a bind operation with a group of 8 registry servers is approximately 2.5 times slower than a group of size 1.

6 Implementation of *FT Unicast*

On the client side of an RMI application, a stub object contains a handle for the remote object that it represents. This handle is represented by the `RemoteRef` interface. The remote reference is used to make method invocations on objects for which it is a reference. The stub object is exported by the server side to the client side. When a client makes a remote method invocation on an object through its stub, first the `newCall()` method of the corresponding remote reference is invoked by giving the remote object name and the operation in the object that needs to be performed. The `newCall()` method initiates a new connection and returns an object of type `RemoteCall` interface. Then, the `invoke()` method of the remote reference is called with this `RemoteCall` object as the parameter to execute the remote method over this connection. The remote method is executed by calling the `executeCall()` method of the `RemoteCall` object.

In order to implement *FT Unicast*, we implemented our own versions of the `RemoteRef` and the `RemoteCall` interfaces. This works as follows. When the `executeCall()` method is called in our implementation of the `RemoteCall` interface, we pass the control of execution to the underlying (and unmodified) RMI mechanism through a method call. If this call is successful, then the result from the remote method invocation is returned to the user. If the call is unsuccessful because of server failure, the existing connection that has been established inside the `RemoteRef` is released and the live reference for this failed server is acquired from the `RemoteRef`. Then, the local RMI registry is contacted first with this live reference to get the name of the server, and then again with the name of the server to get a new live reference for another replicated server. These functionalities are provided by our distributed RMI registry through the implementation

of `reverseLookup()` and `lookupLiveRef()` methods. The latter method, if given a replicated server name as a parameter, randomly returns a live reference for some replica of the server. We do not need to get a complete object reference for the server as we already have a stub for the server. We only need a live reference to establish a connection to another available replica of the server. Then the old live reference (of the now unavailable server) at the `remoteRef` is replaced by the new reference using the `setRef()` method of the corresponding `remoteRef`. Again the process is repeated by making the `RemoteRef` establish a new connection, instantiate a new `RemoteCall` object and then calling the `invoke()` method, until the remote method invocation is successful. Then the results are returned to the client.

The process explained above is executed transparently to the client. That is, the client makes only a single method invocation on a remote object server and if this server is unavailable, the remote reference layer masks this failure by finding an available server, executing the method and eventually returning the results of this method invocation to the client.

As we mentioned earlier, we have our own implementation for the `RemoteRef` interface. Because this handle is exported from the server side, we need to make sure that this handle is correctly bound to our implementation before it is exported from the server side. This is done as follows. In Java RMI, the server object inherits from the `UnicastRemoteObject` which is an extension of a remote server. Instead, we have our own extension that mirrors the `UnicastRemoteObject` which we call the `FTUnicastRemoteObject`. In our case, when the server implementation is instantiated, the `exportObject()` method of the corresponding `FTUnicastRemoteObject` is called. This method instantiates an object of class `FTUnicastServerRef` and calls the `exportObject()` method of this object. This method sets the skeleton to the proper skeleton class, the stub to the proper stub class by setting the `RemoteRef` (which in our case is of type `FTUnicastRef`) correctly, and creates a binding between the remote object and the stub. The stub object is then exported to the client side.

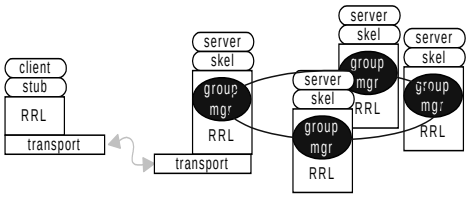

Figure 8: Highly available server architecture

In the next section, we discuss how the `GroupManager` class can be used to build fault-tolerance into any general purpose Java application server through replication.

7 Implementing Highly Available Application Servers

Our implementation of the `FTUnicastRemoteObject` class enables a client to transparently recover from server failures. It works by allowing multiple servers to register under the same name, and in the event of a failure, it redirects a client's request to another server. This method only works however, for *state-less* servers. That is, servers that do not modify their state based on client requests. An HTTP server is an example of a state-less server. State-full servers on the other hand require a general solution that is not provided by *FT Unicast*. In this section we address the general problem by integrating our `GroupManager` and `FTUnicastRemoteObject` class implementations, and provide a general architecture.

The `GroupManager` class has so far been used to construct the highly-available *FT Registry*, a state-full server. This concept can be generalized to make any application server fault-tolerant—by replicating the server and using `GroupManager` class to manage replicas. An architecture for such a highly available server is shown in Figure 8. In this case, the group managers ensure reliable ordering

of events across all the server replicas and guarantee that servers have a consistent state. Failure detection of servers can be performed, as in the *FT Registry*, by the group managers pinging each other in the background. Similarly, dynamic addition of server replicas can be allowed by transferring the state of an existing server to the newly added replica.

The ability to detect server failures, and to transparently redirect a client's request to a replicated server is another key ingredient in our design. We have already demonstrated that this functionality can be integrated within the Java RMI architecture at the RRL level, by implementing `FTUnicastRemoteObject` class. But unlike the `FTUnicastRemoteObject` class, here the client has the illusion of a single server but in reality there are replicated servers that are coordinated by the group managers.

Currently, we are implementing a `FTMulticastRemoteObject` class that can be used in place of the `UnicastRemoteObject` class provided by JDK1.1. The `FTMulticastRemoteObject` class will enable replicated servers to provide the illusion of a single server to a client. A server that inherits this class will become a member of a multicast group and any remote method calls to this server object will be multicast to all the replicas in the group.

8 Conclusions

In this paper we presented the design of *Filterfresh*, a Java package that provides support for building fault-tolerance into replicated Java server objects by implementing an underlying *Group Communication* mechanism. We described the `GroupManager` class that is instantiated with each replica and implements the group communication mechanism. We showed how the `GroupManager` class can be used to construct a fault-tolerant RMI registry server – *FT Registry*. We also described the *FT Unicast* mechanism that enables application server failures to be tolerated at the client stub layer, transparent to the client, using the *FT Registry*. Future work includes completing the implementation of the `FTMuliticastRemoteObject` class that enables the group manager support to be general so that it can be used to make any application server highly available and also extensions to *Filterfresh* to support nested invocations which will be required in this case.

References

[1] G. Beedubail, A. Karmarkar, A. Gurijala, W. Marti, and U. Pooch. An Algorithm for Supporting Fault Tolerant Objects in Distributed Object-Oriented Operating Systems. In *Proc. Fourth International Workshop on Object-Oriented Operating Systems*, 1995.

[2] K. Birman, A. Schiper and P. Stephenson. Light-weight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, August 1991.

[3] K. Birman and R. Van Renesse. *Reliable Distributed Computing with ISIS Toolkit*, IEEE Computer Society Press, 1994.

[4] N. Brown, C. Kindel. Distributed Component Object Model Protocol – DCOM/1.0. *Internet Draft*, 1996.

[5] T. Chandra, V. Hadzilacos and S. Toueg. Impossibility of group membership in asynchronous systems. Technical Report 95-1533, Computer Science Department, Cornell University, August 1995.

[6] D. Dolev, D. Malki, and R. Strong. A Framework for Partitionable Membership Service. In *Proc. of the 15th Annual ACM Symposium on Principles of Distributed Computing*, 1996.

[7] M. Fischer, N. Lynch and M. Peterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, April 1985.

[8] S. Hirano, Y. Yasu, and H. Igarashi. Performance Evaluation of Popular Distributed Object Technologies for Java. In *Proc. of ACM Workshop on Java for High-Performance Network Computing*, 1998.

[9] IONA Technologies. *http://www-usa.iona.com/Press/PR/Isis.html.*

[10] W. Jia. Implementation of a Reliable Multicast Protocol. *Software Practices and Experience*, July 1997.

[11] M. Kaashoek. *Group Communication in Distributed Computer Systems.* Ph.D Thesis, Vrije Universiteit, Netherlands, 1992.

[12] M. Kaashoek, A. Tanenbaum, and K. Verstoep. Using Group Communication to Implement a Fault-Tolerant Directory Service. In *Proc. of the 13th International Conference on Distributed Computing Systems*, 1993.

[13] S. Maffeis. Adding Group Communication and Fault-Tolerance to CORBA. In *Proceeding of USENIX Conference on Object-Oriented Technologies*, June 1995.

[14] S. Maffeis. A Fault-Tolerant CORBA Name Server. In *Proc. of Symposium on Reliable Distributed Systems*, 1996.

[15] S. Maffeis. iBus – The Java Intranet Software Bus. *http://www.softwired.ch/ibus.htm.*

[16] 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*, vol. 39, no. 4, pp. 54–63, April, 1996.

[17] Object Management Group. *The Common Object Request Broker: Architecture and Specification 2.1*, 1997.

[18] F. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, December 1990.

[19] R. Stevens. *UNIX Network Programming*, Prentice Hall, 1990.

[20] Sun Microsystems. *Remote Method Invocation Specification*, 1997. *http://java.sun.com/products/jdk/1.2/docs/guide/rmi/index.html.*

[21] Y. Wang, Y. Huang, K. Vo, E. Chung and C. Kintala. Checkpoint and its applications. In *Proceedings of the 25th IEEE Fault Tolerant Computing Symposium*, June 1995.

[22] A. Wollrath, R. Riggs and J. Waldo. A Distributed Object Model for the Java System. *USENIX Journal*, Fall 1996.