Primary-Backup Object Replications in Java

Li Wang and Wanlei Zhou
School of Computing and Mathematics
Deakin University, Waurn Pond, Vic 3217
Australia
E-mail: {liwang, wanlei}@deakin.edu.au

Abstract

Service replication is a key to providing high availability, fault tolerance, and good performance in distributed systems. Various replication schemes have been proposed, they are based on two streams of techniques, namely passive replication and active replication. This paper focuses on two implementation approaches of the passive primary-backup scheme, remote method invocation approach and replica-proxy approach, using Java RMI and Java network packages respectively. Issues addressed in this paper also include: the primary-backup protocol; restarting a failed server at any site; and a general naming service for the maintenance of dynamic memberships of replica groups. Finally, performance studies based on two implementation approaches are given.
Keywords: Service Replication, Distributed Object, Fault-Tolerance, Remote Method Invocation

1: Introduction

Distributed replication is the maintenance of copies of software and/or data on different sites. It is a common means by which a distributed system can provide continuous services in the presence of failures. Replication schemes generally follow two streams, primary-backup and active replication [2, 4, 5]. In the primary-backup approach, replicas are designated as the primary and backups. Only the primary interacts with clients (processing requests and sending replies), whereas backups stand by and wait for state changes (or updates) from the primary at certain checkpoints. According to how frequently the checkpointing is taken place, primary-backup replications are further categorized into the *hot* scheme (every state change is propagated to backups right away), the *warm* scheme (a set of state changes are propagated at checkpoints), and the *cold* scheme (no propagation) [1]. Primary-backup schemes are called passive replication for the reason that backups passively receive state changes without having any interaction with clients.

In the active replication, all replicas interact with clients. A query request is handled by one replica, an update request is dispersed to all replicas. In contrast to the passive replication, it is named active for the reason that all replicas are actively taking clients' requests. Active replication is also referred as the *state machine approach* [6]. A state machine is used to model each replica as a deterministic process. The consistency between replicas is preserved by delivering the same sequence of ordered operations to each replica.

Most of the research on primary-backup replication [2, 4] is concentrated on consistency protocols of employing multiple backups. It is argued though, if we can restart a failed replica (primary or backup), there is no need to employ a collection of backups, provided the primary and the backup do not fail at the same time. Thus, we demonstrate one backup approach in this paper.

The aim of this paper is on two implementations of primary and backup objects in Java, *remote invocation method (RMI)* approach and *replica-proxy* approach. The RMI approach uses the Java RMI [7] tool as a vehicle to implement primary and backup objects. It is a simpler and faster implementation approach, but less flexible. Whereas the replica-proxy approach based purely on Java network packages is a slower implementation, however, brings a better performance to final systems.

The rest of the paper is organized as follows. Section 2 presents the system model containing the primary-backup protocol, failure semantics, and a naming service. Section 3 describes two implementation approaches in detail. Section 4 studies the performance. Section 5 concludes the

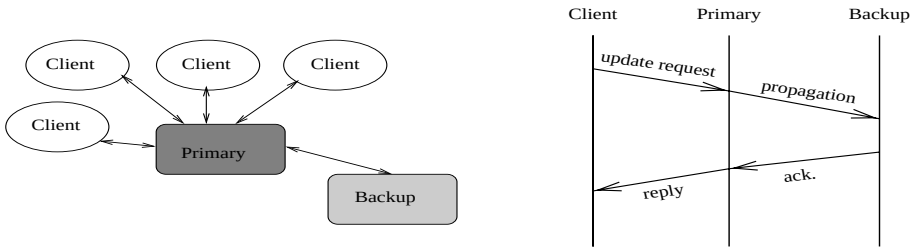

Figure 1. (a) **The Primary-Backup Model.** (b) **The Primary-Backup Protocol**

paper.

2: System model

As depicted by figure 1(a), there are three types of software entities involved, namely clients, the primary, and the backup. Each entity is constructed by an integration of objects. Basically, an object is composed of some state variables and is encapsulated by a set of operations. The state variables can only be accessed exclusively by the set of operations. Operations are categorized as either queries or updates. A query does not change the state of the object but an update does.

Choosing the right checkpointing policy (*hot*, or *warm*, or *cold*) depends on the environment in which the replication system is running, and the requirement on the fail-over time which is the time period from the primary's breakdown to the backup taking over. Obviously, the hot policy gives the shorest fail-over time.

2.1: The primary-backup protocol

We chose to use the hot policy so that the primary-backup system can be recovered without any delay. Figure 1(b) shows the primary-backup protocol.

- *Client side.* Clients send requests only to the primary.
- *Primary side.* The primary propagates each update request to the backup. After the backup returns the acknowledgement, the primary replies to the client.
- *Backup side.* Upon receiving a propagation from the primary, the backup executes the request and acknowledges.
- *Unique primary at any time.* Both the primary and the backup agree that there is a unique primary at any time.
- *Replica consistency.* The primary and the backup executes the same set of operations in the same order.

2.2: Failure semantics

A failure is defined by either a process failure or a site failure. The process failure implies the process has terminated by an exit or by a crash. The site failure implies the host in which a replicated server is running has crashed. The process failure can be detected by a hint from the underlying operation system, whereas the site failure can only be suspected using time-outs.

In the primary-backup replication model, the primary monitors the backup. Upon detecting the backup's failure, the primary disconnects with the backup and will not propagate any subsequent requests. Those subsequent requests are saved into a log file and transferred to the backup when the backup is restarted. Clients monitor the primary. Upon suspecting the primary's failure, clients switch to the backup and re-sends their last request. Duplicated requests from clients can be filtered out by attaching sequence numbers to requests. The following gives the life-cycle of both primary and backup objects:

- *The life-cycle of the primary.* It is started. → It fails. → It is restarted as the backup. → It joins the primary. → It becomes the new primary when the primary dies.
- *The life-cycle of the backup.* It is started. → It joins the primary as the backup. → It becomes the new primary when the primary fails; Or it is started. → It fails. → It is restarted as the backup.

2.3: Naming service

In practice, the failed primary or backup are restarted as soon as possible. However, when a failed one is restarted at a different site, this new site has to be passed to clients. There are generally two methods to solve this problem. First, when the backup is restarted at a new site and joins the primary, the primary piggybacks the new reference to clients. Second, a naming service (NS) is set up separately on a stable site to manage the site changes. We chose to use the NS approach to deal with site changes as it can be extended to a general service provider for maintaining multiple replication groups. The NS interface is then defined as follows:

```
public interface NamingService {
  void createGroup(String[] members, String groupname);
  void addMember(String groupname, String member);
  void deleteMember(String groupname, String member);
  String bind(String groupname);
}
class NSImpl implements NamingService { }
```

When a replica group is created, the coordinator of the group registers the group reference with the NS (by invoking `createGroup()`). The coordinator is also responsible for updating the NS with any member changes (by invoking `addMember()` or `deleteMember()`). For the primary-backup scheme, the primary is the coordinator, which invokes `deleteMember(thegroup, backup)` upon the backup's failure, and `addMember(thegroup, backup)` upon the backup's joining. Whereas clients use `bind(group)` to get the reference to the primary.

3: Implementation approaches

This section discusses the implementation detail based on two approaches, RMI approach and Replica-proxy approach. Both primary and backup execute the same code fragments but are started differently as being either the primary or the backup. Implementation approaches follow the protocol depicted by figure 1(b).

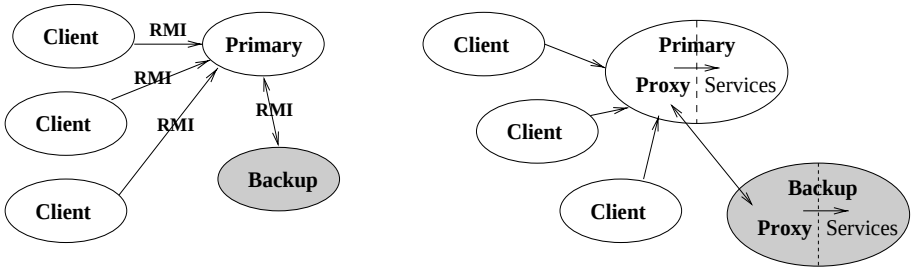

Figure 2. (a)**RMI Approach.** (b) **Replica-Proxy Approach**

3.1: The Java RMI implementation

A RMI (the same concept as remote procedure call (RPC)) system generally provides a separate IDL language to define the interface for a remote service. Using a RMI system to implement primary and backup objects implies clients making RMI calls to the primary, and the primary making RMI calls to propagate client calls to the backup. Figure 2(a) depicts this approach.

Java RMI does not introduce a separate IDL language, rather it introduces RMI packages for the implementations of remote objects. Thus, communications among clients, the primary, and the backup are via three RMI interfaces: the service interface, the primary-backup interface, and the replica interface. We use a service `RoomBooking` as an example. Two operations are provided: `book()` is an update operation, and `booking()` is a query operation. The interface between the primary and the backup has three methods: (a) `join()` is used by the backup to join the primary; (b) `stateTransfer()` is used to transfer the state of the primary to the backup at the instant of joining; (c) When the primary or the backup suspecting the failure of the other, it uses a `kill()` to confirm the decision. The killed one has to be restarted and then rejoins the primary. The following code fragment shows these interfaces:

```
public interface RoomBooking extends Remote {
  public boolean book (String booking);
  public String  allBookings (Date D);
}
public interface PrimBackup extends Remote {
  void join (String backup);
  void stateTransfer();
  void kill();
}
public interface Replica extends RoomBooking, PrimBackup { };
```

The primary-backup system can be started in a couple of ways: (1) The primary is started first, the backup is started with the reference to the primary and joins the primary. (2) The backup is started first, the primary is started with the reference to the backup and the primary sets up the connection to the backup.

3.2: The Replica-Proxy implementation

This approach builds up the replication system based on networking packages `java.net` instead of the RMI system. Replica proxies are used as replica stubs that service objects can plug in and play. In turn, replicas are an integration of two kinds of objects, *service objects* such as `RoomBooking` service, and *replica-proxy objects*. A service object provides services without being involved in matters like where it should be located or whether it is replicated. The replica proxies deal with network connections, message marshallings, and more importantly replication managements. Figure 2(b) depicts this approach.

The proxy object is designed by using: (1) A dedicated communication channel is set up between the primary and the backup. (2) A parent thread is used to accept all clients' connections at any time. Upon each connection, the parent thread spawns a child thread to deal with that client. (3) A thread is started by the primary to accept the backup so that the backup can join at any time.

4: Performance studies

The performance study is conducted within a local subnet (10M Ethernet) of Sun sparc stations under Java JDK 1.1.2 version. The tests use one client making synchronous (blocking) requests to the primary. The tests are performed by measuring one round-trip time of sending a request and receiving different sized replies. In figure 3, *X-axis* represents the reply message sizes (in Kbytes), whereas *Y-axis* represents the response time (in msec) of average 100 requests. Two groups of experiments are performed based on two implementation approaches. Both are based on two settings: (1) a single primary; (2) the primary and the backup. Figure 3 shows the test results:

1. With the RMI approach, depicted by lines $RMI-P$(single primary) and $RMI-PB$(primary and backup), the tests show there is some performance penalty involved by introducing the backup. This is largely because two RMI calls are issued before replies are sent back to clients.
2. With the replica-proxy approach, the tests show that introducing the backup does not bring a significant performance penalty. The two lines, Proxy-P(single primary) and Proxy-PB(primary and backup), are very close to each other and crossed sometimes. Also we can

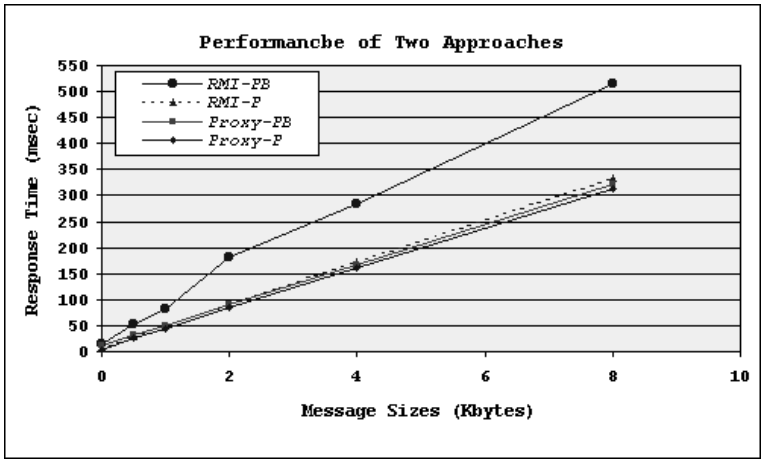

Figure 3. The Performance of RMI and Replica-Proxy Implementations in Java

observe the replica-proxy approach is better performed than the RMI approach. This is because of the optimized codes in the replica-proxy implementation, in particular, the backup only sends acknowledgements to the primary instead of request results.

5: Remarks

In this paper we have presented the design, two implementation approaches, and the performance studies of the primary-backup replication scheme. Generally speaking, the RMI approach is a faster, but less flexible implementation approach. The replica-proxy approach is a slower process, but results in better performance and optimized codes, especially, performance can be better achieved when replies involve a large volume of data. More complex replication schemes can be better implemented using the replica-proxy approach, which can serve as a general design pattern for any object replication system, such as the active replication schemes.

We also introduced a naming service for maintaining dynamic replication groups so that a failed replica can be restarted at any site. In practice, however, the failed one is often restarted at its original site.

Java, as a fully object-oriented, platform-independent, and Internet-friendly language, has gained enormous recognition [3]. Through the experience of authors, it is indeed proven a neat and easy-to-work-with high-level language for developing fault-tolerant and distributed programs.

References

[1] A.J.Wellings and A. Burns. Programming replicated systems in ada 95. *The Computer Journal*, 39(5):361–373, 1996.

[2] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. The Primary-Backup Approach. In Sape Mullender, editor, *Distributed Systems*. Addison-Wesley Publishing Company, second edition, 1993.

[3] James Gosling. The Feel of Java. *IEEE Software*, pages 53–57, June 1997.

[4] Rachid Guerraoui and Andre Schiper. Software-Based Replication for Fault Tolerance. *IEEE Computer*, pages 68–74, April 1997.

[5] David Powell and Paulo Verissimo. *Delta-4: A Generic Architecture for Dependable Distributed Computing*, chapter 6, Distributed Fault-Tolerance. Springer-Verlag, 1991.

[6] Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, pages 299–319, December 1990.

[7] Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, California 94043. *RMI Specification*, JDK 1.1.2 edition, 1997.