An Architecture for Object Replication in Distributed Systems

Ganesha Beedubail and Udo Pooch

{ganeshb, pooch}@cs.tamu.edu

Department of Computer Science, Texas A&M University,

College Station, TX - 77843.

Technical Report (TR_96-006)

March 1996.

Abstract

In this paper we present an architecture for developing replicated objects in object oriented distributed systems. The architecture is however general and the concepts can be adapted to non-object oriented distributed systems. The architecture specifies the main requirements for developing replicated objects in distributed systems, namely: the replicated object representation, the replicated object development and management mechanism, and the replica consistency protocol. For achieving these requirements the architecture specifies a hierarchy of objects and the inheritance and functional relationship among these objects. All the management functions of replicated objects are encapsulated in a manager object. An application object inherits these functionalities from the manager object. This object replication architecture will greatly reduce the complexity of the development of replicated objects in distributed systems. We also developed an efficient replica consistency algorithm that takes advantage of the features of our architecture. A preliminary prototype implementation of our architecture has been carried out in Spring OS environment.

1 Introduction

The advent of inexpensive micro computers, powerful workstations and high speed local area networks made the distributed systems ubiquitous. In simple terms, a *distributed system* is a collection of computing nodes interconnected by a communication network, in which software components run on multiple nodes and communicate over the network. An ideal distributed system appears as a single system to the user. This *single system image* is provided by the software components that execute on the system. The cost effective distributed systems are rapidly replacing the traditional centralized (mainframe based) systems.

As distributed systems become common place and support *mission critical* application systems, the reliability of these systems becomes a crucial issue. When we talk about reliability of distributed systems two contradicting facts become apparent. Since distributed systems are made up of many components, there is no single point of failure. But since there are more number of components, the probability of failure of any single component also increases. When the system developers provide transparent system services in a distributed system, unless special care has been taken, there is a higher chance of service failure. However regular system developers will neither have the time nor expertise to develop fault tolerant services (objects[1]).

In this paper we will present a solution for providing/developing *highly available (fault tolerant)* objects (services) in distributed systems. The technique we use for this purpose is the *replication of objects* on system components that fail independently. The concept of replicating objects for providing high availability is not new. However realizing a replicated service in practice is quite hard. The reason is simple. When developing a replicated object, the developer has to consider difficult facts such as *partial failures* and *concurrent activities* that are inherently present in any distributed system. The solution we provide here is a software architecture that greatly reduces the complexity of the development of replicated objects in distributed systems.

Much of the research[1, 2, 3, 4, 5] in object replication techniques is concentrated on one aspect of replication, i.e., the *replica consistency*. A few systems[6, 7, 8] elaborate on other aspects of replication such as *naming support* and *replicated object management* (creating, adding, deleting replicated objects). In this paper we will present a comprehensive architecture for developing replicated objects in distributed systems. This architecture provides the mechanisms and guidelines for the object developer to develop replicated object. In other words this is a *frame work* for object replication that can be tailored for individual needs with minimal effort. Our architecture encapsulates all the functions necessary to manage the replicated objects in a *manager object*. An

[1] In this paper the terms service and object are used interchangeably.

application object can inherit these functions from the manager object. We also developed an efficient replica consistency protocol for this architecture.

The rest of the paper is organized as follows. The next section describes the related work. It also presents our design goals and our system model. Section 3 presents our architecture for object replication in object oriented distributed systems. Section 4 discusses the replica consistency protocols for our architecture and presents our replica consistency protocol. In section 5 we will discuss the issues related to the naming service for supporting object replication. Section 6 discusses the concurrency and load sharing issues in object replication. Section 7 describes our implementation environment and our prototype implementation. Section 8 concludes this paper.

2 Related Work

Object replication is a method of providing fault tolerance, usually used to provide highly available services. Here multiple copies of the object are executed on separate machines that fail independently. Thus the probability of survival of the object as a whole is greater than the probability of the survival of any individual copy of the object. The probability of survival of the object can be varied by varying the degree of replication. Another advantage of the replication technique is that, the multiple servers can share the application load (if the application semantics allows this). If the majority of the requests to the server is *read only*, then the replication technique is really attractive.

Replicated (fault tolerant) objects are generally structured in one of the following two ways (or variations of these): 1) Replicate the service state at all the servers (machines) and present the client requests in the same order to all the non-faulty servers. This approach is known as *active replication* or the *state machine* approach[4]. 2) Designate one server as the primary and all others as backups. Clients make requests by sending messages only to the primary. The primary forwards the requests to the backups (so that, backups can synchronize their state with the primary) and then responds to the client[2]. If the primary fails, then one of the backups becomes the primary. This approach is known as the *primary-backup* or the *primary copy* approach[1]. All the research on object replication are variations or combinations of the above two basic strategies.

Alsberg and Day [1] proposed one of the earliest primary-backup protocols. The paper presents the error detection and recovery schemes for *two-host resiliency* (tolerating one failure) and the extension of this concept to n hosts. In [5], Budhiraja et. al. present some theoretical aspects of the primary backup approach. Their system assumes closely synchronized clocks in the system. They classify the primary-backup algorithms as *blocking* and *non-blocking* and provide lower bounds

[2]The algorithms vary in how the request is forwarded to the backups and when the primary responds to the client.

(under various failure conditions) on the *degree of replication, blocking time* and *failover time* for any primary-backup algorithm. The Harp File System also uses a primary-backup protocol[9]. It is assumed that the system clocks are closely synchronized and all the machines are equipped with an uninterruptable power supply (UPS). The protocol keeps a log of updates on volatile memory (the UPS is necessary to write back this log to disk in case of power failure). Operations on the log are applied to the system in the background. Thus the disk access is removed from the critical path and good response time is achieved. In [10], Beedubail et. al. extended the primary backup protocol for *nested object invocation* (i.e., to service a client request object can invoke the services of other objects). That scheme assumes the existence of a reliable multicast and fault tolerant naming service.

The *replicated distributed programs* described by Cooper[6] is an example of active replication. Here, all the replicas (called the members of a *troupe*) execute the client calls independently. A higher level protocol should make sure that all the replicas get their client calls in the same order. In [11] Birman et. al. describes how *totally ordered, reliable broadcasts* can be used to maintain *replicated data* (active replication). In [7] Olsen et.al. describe the *interface group* abstraction developed and implemented on ANSA architecture. *Interface groups* hide the group membership from the application and thus can be used for supporting reliability by replication.

Object replication in Arjuna[8] uses active replication[3]. The replication algorithm uses the support of *naming and binding* service and the *atomic action service* provided by the Arjuna system. A higher level group locking protocol (issued by the application program) avoids the need for a reliable ordered broadcast. The clouds system[12] uses the concept of *parallel execution threads (PETs)* to support replicated computation (and thus fault tolerance). In [13] Kaashoek et.al., describe the *fault tolerant directory service* implemented in the Amoeba distributed operating system. They use the *closed group communication* service supported in Amoeba to implement the directory service. In [14], Spiewak describes a framework for replicating objects in Spring OS using primary backup algorithm. Here the replicas are periodically updated. Thus the replicas will not always be in a mutually consistent state. The reader can refer to [15, 16, 17, 18] for additional reading on replication.

2.1 Motivations for this Work

The foregoing discussion mainly dealt with various algorithms for object replication. These algorithms basically keep the copies of replicated objects in a mutually consistent state. However we are interested in building a complete architecture for developing replicated objects in distributed

[3]Arjuna also uses single copy passive replication, similar to coordinator cohort scheme.

systems. We believe that the task of providing replicated objects in distributed systems should address the following issues:

1. **Object Representation:** It is necessary to have a representation mechanism for replicated objects.

2. **Object Development and Control Mechanism:** We need mechanisms to develop the replicated object, name the replicated object, start a replicated object, add a new member to the replicated object, delete an existing member from the replicated object and migrate the replica from one node to another. We call this *the server development framework.*

3. **Replica Consistency Protocol:** The protocol that keeps the states of replica in a mutually consistent state. This may also specify how the client accesses the replicated objects.

All of the above issues are not independent of each other. However, it is useful to identify these as the main issues in providing replicated objects since we need to provide solutions to all of these issues when providing any practical replicated service. The existing systems either do not address all these issues or the solutions provided are complex and expensive. Our research work provides better solutions to these issues. We provide an architecture for developing replicated objects. In addition we also developed a replica consistency algorithm that makes use of this architecture. The algorithm is less expensive compared to the existing replica consistency algorithms.

Some other issues that we considered are : *application transparency* and *ease of development.* Most of the research work strives to provide a replication algorithm, that provides an application transparent replication facility (i.e, the application programmer need not know that the object will be replicated). But we believe that, the object replication is a complex issue and leaving the replication management to a general purpose tool may degrade the system performance to an unacceptable level. Thus we decided that the server development process not be transparent to the replication. The developer knows that object will be replicated. The architecture provides a base for implementing various replica consistency protocols. Thus the developer can make use of the application knowledge while implementing the replica consistency algorithm. Depending on the application semantics the developer can relax the replica consistency requirements and can get an optimum performance. However, the architecture provides replica transparency at the client side. The client is unaware of the fact that the object is replicated. Thus it accesses the replicated and non-replicated objects in a uniform fashion. The modular nature of our architecture makes it very easy to use it for the development of replicated objects. The developer uses the *off-the-shelf* object development environment and remote method invocation facility for the communication. We do not use any customized low level message passing for reliability.

2.2 System Model

Our system, for which we developed the object replication architecture is a distributed object oriented system. This system is a collection of objects that provides services to the user. The objects interact with each other and with the outside world through well defined interfaces. The object interfaces are defined in such a system using some standard interface definition language (IDL). Systems such as SPRING[19] and CORBA framework[20] provide such features. The objects in the system are active (i.e. the object is always executing for providing services). We will not consider *persistent objects* here (note that *object persistence* is an *orthogonal* issue with respect to object replication). The following points summarize our assumptions for the rest of this paper.

1. We assume an object oriented distributed system. The inter-object communication is carried out using remote procedure invocation.
2. The processes in the system observe *fail-stop* behavior. The *network-partition* failures are rare. These assumptions are only necessary for the correctness of replica consistency protocol.
3. The system supports a distributed naming service (NS). The NS supported in the system is reliable.
4. When objects are ready to offer a service (say when they are started up) they export their interface to the NS (i.e., the object reference is bound in the name server with a name).
5. When objects want to invoke the methods (operations) of other objects, they import the interface for that object form the NS (i.e., they get the object reference from the NS). Usually an object which invokes the services of another object is termed the *client* and the invoked object is termed the *server*.

3 Architecture for Object Replication

In section 2.1 we identified the issues that are to be addressed while providing support for the development of replicated objects. Here we will present our solutions for these issues.

3.1 Object Representation

We represent a replicated object by an array of object references. The object reference is like a capability. If we have an object reference, we can access the object without knowing the location of the object (however the object reference would encapsulate the location of the object inside itself). We call this array, the *glist* (for group list). The NS associates the name of the replicated object with this *glist* . Thus when a client resolves a name that represents a replicated object, it gets the *glist* representing the replicated object. Note that this fact is transparent to the client; the client need not know whether it gets the object reference for a single object or a *glist* when it resolves a name. If the name represents a replicated object, then the NS automatically creates a *proxy object* to handle the invocation of the replicated object. This is done by the name service library routines linked with the client load module (this is explained later in section 3.4.2).

3.2 Object Development and Control Mechanism

Developing a replicated object from scratch is a very involved task. Thus we provide a standard abstract object which we call the *manager object (the Mobj)* and a set of standard development methodologies to develop a replicated object. The standard functions such as *addition* and *deletion* of members to/from replica set are provided by the *Mobj* . The methods in *Mobj* provides a default behavior (which is adequate in most of the cases), but can be overrided by the developer for special needs. The development procedure specified in this architecture can be fully automated with the help of a language preprocessor and slight modification to the IDL compiler. Section 3.4 presents the details of the architecture for the development and control of replicated objects.

3.3 Replica Consistency Protocol

Our architecture does not restrict the replica consistency protocol. In fact it is quite general since our objective here is to provide a base for implementing a variety of replica consistency protocols. One can implement any replica consistency protocol that is suitable for the application at hand. However, we also developed a replica consistency protocol that exploits specific features of our architecture and thus is very efficient compared to the protocols found in the literature. Section 4 provides a discussion on implementing replica consistency protocols in this architecture and the protocol developed by us.

3.4 Procedure for Developing Replicated Objects

In this section we will briefly describe the procedure for developing replicated objects in distributed systems using our architecture. Before going into the details of the development of replicated objects, we will briefly review the typical development procedure for a non-replicated object in a distributed system. This is important because the development procedure for replicated objects is a modification of this process.

3.4.1 Development of non-replicated objects

Figure 1 shows the steps involved in the development of a non-replicated object in a distributed system. Assume that the programmer wants to develop an object called *sobj* (for some object). Typically the programmer will start with defining an *interface definition* file (say *sobj.idl*). This

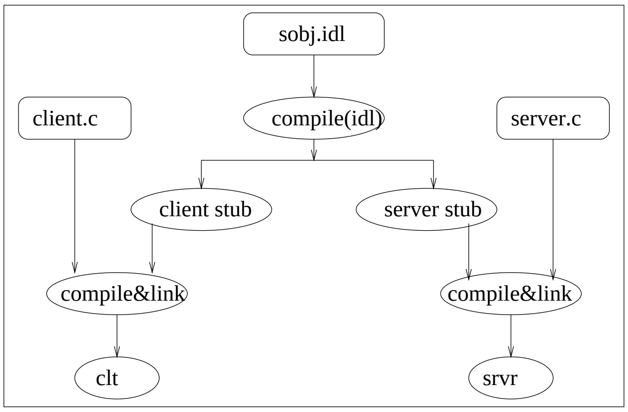

Figure 1: Development steps for non-replicated objects.

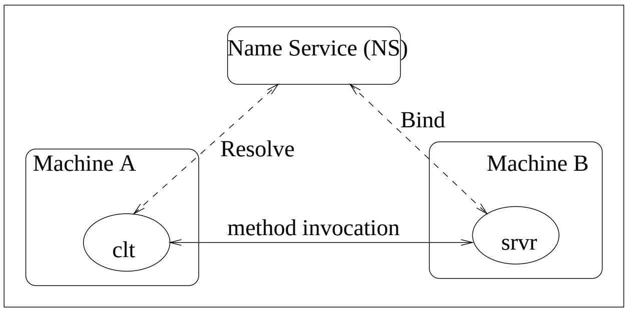

Figure 2: Execution logic for non-replicated objects.

interface definition describes the methods/services of the object. Then the programmer will implement the application logic for the object in a high-level language (say in file *server.c* using 'C' language). The programmer may also implement the application logic for the client of the object (in *client.c*). Then the *sobj.idl* file is compiled by an *IDL compiler* which generates the server and client *stubs*. These stubs are later compiled and linked along with the application server and client codes respectively to generate the final executable modules.

Figure 2 shows the execution logic for non-replicated objects. Typically the server object is started either automatically when the system is booted up or by a system administrator. The server exports its interface to the NS (i.e., *Bind*s its name in the NS, thus indicating that it is available for service) and will wait for the service request. Later when a client requires the service of this object, it will import the object interface (i.e., *Resolve*s the object name from NS and gets the object reference). Using this interface (object reference) the client will invoke the relevant method of the object.

3.4.2 Development of replicated objects

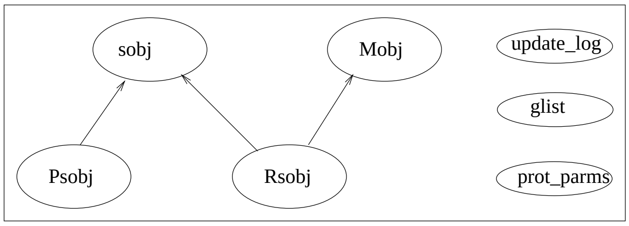

Figure 3: Object hierarchy involved in developing replicated objects.

Assume that we need to develop a replicated version of *Sobj* . Figure 3 shows the objects involved in the development of replicated *Sobj* . The appendix A shows the complete list of methods and variables associated with these objects. Here we will briefly explain the role of each object found in the figure and the rationale for using them.

Sobj: This is the original object that we want to replicate.

Mobj: We call this the *manager object* in our architecture. This provides the common interface that is necessary for replicating and managing any object (i.e., it provides the management interface for a replicated object).

Rsobj: This is the replicated version of *Sobj* . This object inherits from *Mobj* and *Sobj* . By inheriting from *Mobj* , *Rsobj* gets all the features (or interfaces) necessary for acting as a replicated object. Inheriting from *Sobj* makes *Rsobj* a subtype of *Sobj* . Hence *Rsobj* can be *bound* in NS as an instance of *Sobj* . Thus the clients can resolve *Rsobj* as an instance of *Sobj* , providing *replication transparency* for the client. This object also will have a modified update method for each update methods of *Sobj* . This modified method will have a parameter of the type *prot_parms* (see below) along with all the original parameters.

Psobj: This is the *proxy object*. It is used to affect the replication transparency at the clients. It inherits from *Sobj* . The purpose of this inheritance is to obtain the method signatures of *Sobj* for *Psobj* . The methods of *Psobj* implement the *replica access* (replica consistency protocol).

update_log: This object stores the information related to the update of the application object. It stores the parameters for an update and the results of that update. This information is used for tolerating failures.

glist: This contains the list of object references for all the replicas of a replicated object. The glist is associated with every replica and with the *replica name* in the NS.

prot_parms: This is a structure that consolidates the parameters required to implement the replica consistency protocol.

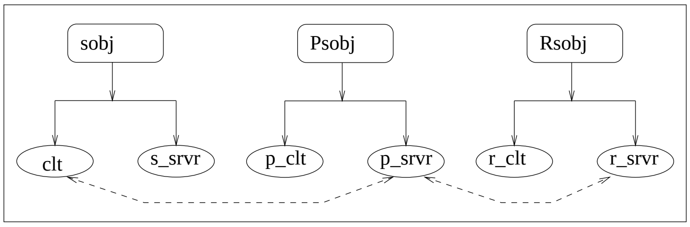

Figure 4: Development steps for replicated objects.

After providing the above background, we are ready to explain the development process for replicated object. First, we will explain the development procedure without assuming any help from IDL compiler. Later we describe how the whole process can be automated by modifying the IDL compiler.

First, the programmer writes three object interfaces *Sobj* , *Psobj* and *Rsobj* as specified above. Next the programmer writes the client and server implementations for all the interfaces. The usual compilation generates the appropriate executables. Figure 4 shows this process. For clarity we did not show the client and server codes. Note that it is not necessary to write the client implementations for *Psobj* and *Rsobj* . In the following we will briefly explain the logic contained in each module.

r_srvr: This contains the logic for the server part of the replica consistency protocol and the actual application logic. The replica management logic is inherited from *Mobj* . The application logic can be easily implemented by instantiating an instance of *Sobj* object. This object has an instance of *glist* and *update_log* objects for implementing replica consistency protocol. Note that for every update method in *Sobj* , we have a corresponding modified update method which has just one additional parameter (of the type *prot_parms*). Thus this method can be properly used to implement the appropriate replica consistency protocol.

p_srvr: This implements the client part of the replica consistency protocol. It uses the client stubs of *Rsobj* to access the methods of *r_srvr*. This too will have an instance of *glist* for implementing replica consistency protocol.

s_srvr, clt: These will implement the actual application server and client logic respectively.

p_clt, r_clt: These modules need not be implemented.

Figure 5 shows a typical execution of a replicated object. When a replica (i.e., *r_srvr*) is started either manually or automatically, it forms a replica group, if it is the first replica or adds itself (i.e., joins) to the replica group if there is already a replica group (the details given in section 4). As a side effect, the NS gets updated to reflect the latest *glist* that contains the references to all

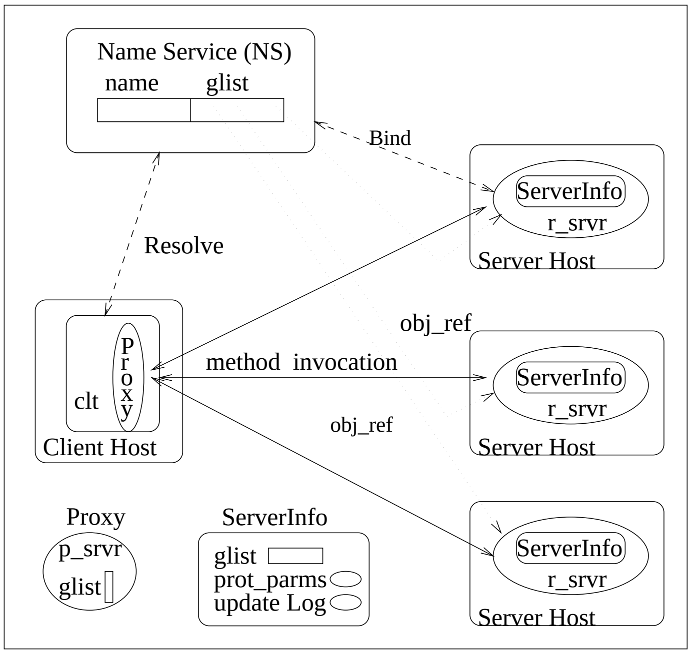

Figure 5: Execution logic for replicated objects.

the replicas. When a client wants to access the (replicated) object, it *resolves* the name in NS (using the *resolve* method of the NS object). The NS knows that the object is replicated. Thus instead of handing the client the object reference of the server, it creates a *proxy object* (*p_srvr*) at the client's domain (address space) and hands over the reference of the proxy object to the client. Since the *Psobj* inherits from the *Sobj* , the client can use the proxy reference similar to the non replicated object reference. Now, the client's invocation invokes appropriate method at the proxy object. The proxy knows the existence of replicas and contacts the replica (according to the proper replica control protocol) and gets the results. These results will be given back to the client. Observe, how the replication became transparent to the client in this case. Had there been a non replicated object, then also the client would have carried out the similar invocation on the object reference handed to it by the NS. But in this case, however, the NS would not have created the proxy object. Now take a look back at Figures 1 and 4. In the case of non replicated object, the *clt* invocation goes to the *s_srvr*. In the case of replicated object, the *clt* invocation goes to *p_srvr* and *p_srvr* invokes the *r_srvr*.

3.5 Automation of Development Procedure

From the Figure 3 and the details given in the appendix A we can observe the following. Once the application object *Sobj* is given/specified then the specification of all the other objects are fixed. Moreover, we can derive these specifications mechanically (i.e., without any semantic knowledge of the application). Thus a computer program will be able to generate all these objects for us. The

IDL compiler is best suited to do this job. We can develop a language preprocessor such that it generates all the objects shown in Figure 3 given the *Sobj* . If it is acceptable for the developer, then we can also fix the replica consistency protocol; the code for this can also be generated by the compiler (appendix B shows the logic for one such algorithm[4]). In this case the developer just needs to implement the application logic without bothering about the complexity of the replication logic. However we have not yet attempted to automate this process yet.

4 Replica Consistency Protocol

In this section we will describe how our architecture provides the base for implementation of any replica consistency protocol. Usually replica consistency protocols found in the literature needs the following variables: *(a) the object references, (b) protocol specific variables, and (c) a log storage to store the messages (for retransmission).* Our architecture provides all these in a modular fashion. That is, we provide the *glist* (list of objects in the replica group), *update_log* (to store the updates) and the structure *prot_parms* that can be used to pass the protocol parameters along with the invocation. Note that the *glist* is available at all the replicas in addition to the clients and the NS.

In the following we will briefly describe our replica consistency algorithm. Since the replica management functions (like replica addition/deletion) can affect the replica consistency, we will also describe the logic of replica management functions[5]. The explanation is brief here. The pseudo code and sketch of proof for the algorithms is given in the appendix B.

Our replica consistency algorithm is tailored to the architecture. This algorithm is a variation of the *primary backup (or master slave)* algorithm. The novelty of this algorithm is : (1) It just uses one round of message passing (*one phase protocol*) where as the existing protocols use a minimum of two rounds of message passing (*two phase protocol*). (2) It is self contained. The group membership and failure detection are seamlessly integrated into the protocol. (3) It is specified and implemented using only RPCs. Note that this feature is useful in our architecture since we are using the object oriented framework where the inter-object communication is specified in terms of remote method invocation (synchronous communication). When we use this feature, the programmer need not be concerned with the low level message passing details.

[4]In fact we can fix some standard replica consistency protocols that may be applicable for various situations and let the compiler incorporate one of them to the code as requested by the application developer.

[5]The algorithms for deleting a replica and listing the current replicas are straight forward and thus not described here.

4.1 The Master-Slave Replica Consistency Algorithm

In our algorithm, one replica is designated as the master and all the other replicas are slaves. The client (here the client means the proxy object specified in Section 3) invokes the method on the master. The master forwards this request to the slaves and then returns the result to the client. The master, slaves, and the client have the *glist* (that contains the object references of all the replicas). The invariant property that is maintained (guaranteed) by the algorithm is that:

- Everyone believes that, the first machine in the glist (whose status is UP) is the current master. This belief may be incorrect. If it is incorrect, it will not introduce any inconsistency in the system.

Initially, when a client wants to access the replicated object, it obtains the *glist* from the NS. It then sends the invocation to the first object in the *glist* that is *alive*. If that object happens to be the master the client gets the result for the invocation. Otherwise, the object returns the object reference for the current master. The client then contacts this object. Note that it may be possible that all the object references in the client's *glist* are stale at some point in time (i.e., the object references do not point to a live object). Then the client gets a fresh *glist* from the NS.

Now we will see what happens at the server object side when a server object receives an invocation. The server object maintains two variables along with its local state: (1)**GUN** (global update number): This variable gets incremented every time an update is made to the object state. (2) **Master_flag**: This flag indicates whether the object is the master or the slave. When an invocation arrives at the object (let it be O_i), the following two cases are possible.

(1) The object (O_i) is the master: In this case the object (O_i) performs the local update, increments the GUN, updates the *update_log* and forwards the invocation to the slaves. Finally it returns the result to the client. Note that if the invocation is *read-only*, then O_i simply returns the result without performing any updates or without forwarding the invocation to the slaves.

(2) The object (O_i) is not the master (this can happen when the client did not have a correct reference to the current master object) : Then the object (O_i) *"ping"*s (sends *are_you_ok* message) all object replicas whose object references appear before O_i's object reference in object O_i's *glist* (in a left-to-right order). If O_i receives an *ok* reply from an object O_j then the object O_j is the current master. Object O_i returns the object reference of O_j to the client. If O_i does not receive any *ok* reply, then all those objects are suspected as failed. This implies O_i should become the master. Thus O_i assumes the mastership. While assuming the mastership, O_i forwards its last update to all other slaves so that all the replicas are consistent at this point. Next O_i performs the

logic given for case (1).

4.2 Algorithm for Adding a Replica

Here we will describe how to add a replica to the replica group. If there is no replica group (i.e., replicated object is not started) then this algorithm will create a replica group. The same algorithm will work for an object that is recovering from a machine failure and wants to join the group again.

When the object (say O_i) is started, it resolves the replica name (and thus gets the *glist*). If the *glist* is empty, then O_i is the first replica. It updates the *glist* (with its object reference) and binds it in the NS. If the *glist* is not empty, then O_i needs to join the existing replica group. In this case object O_i follows the master-slave protocol for joining the group (the previous section explains this protocol). However, as a side effect, the current state (and updated *glist*) of the replicated object is transferred to the object O_i and the NS is updated with the new *glist*.

5 Naming Service for Object Replication Architecture

In the previous sections we assumed the existence of a NS. We also required that the NS be fault tolerant and provide support for object replication. We developed such a NS. Here we will briefly highlight the features of our NS. The details of this work can be found in [21].

We used Spring OS for our implementation[19]. Spring provides a native name service[22]. To make it fault tolerant, we replicated the Spring NS. The main idea of the replication algorithm is similar to the one given in Section 4. However, the details are somewhat different. First, we fix the name server machines (NSmachines) in the system apriori. All the machines in the system (network) know the identity of these NSmachines (through a configuration file). Thus when a name server client in any machine wants to access the NS, it would have the object reference of all the replicas of the name server in the system. Thus it can access the replicas according to the replica consistency protocol.

Note that the static configuration file is not a limitation. Since NS is a stable basic service, rarely do we need to change the NSmachines. If we need to change the NSmachines then the system administrator can do it manually. Also note that for a basic service like NS there are only two ways of locating the servers: (1) by static assignment, (2) by broadcasting. We used the static assignment scheme since we assumed a *point-to-point* communication link. Also note that in the current LANs broadcast is not possible across a subnetwork.

We also modified the *resolve* method of the Spring NS to support our object replication framework's requirement. During the resolve operation the NS does the following: if the resolved object is non-replicated then the NS simply returns the object reference of this object to the client. If the resolved object is a replicated object, then the NS creates a proxy to access the replicated object and returns the object reference of the proxy to the client.

6 Concurrency Control and Load Sharing

Concurrency Control: Replication of objects introduce a major problem in the concurrency control area. We assume that the objects we want to replicate are multi-threaded, i.e., they service multiple clients concurrently (multiple threads of control are active concurrently inside the object). In this case the object becomes non-deterministic as explained below.

An object is said to be deterministic, if in two or more executions, starting from the same initial state and servicing the same requests in the same order, the object reaches the same final state. When objects are multi-threaded, the serialization order for two different executions need not be the same. Thus the final state after two executions (that received same requests in the same order) need not be the same.

Thus when we replicate these (multi-threaded) objects, we can not guarantee the replica consistency unless we use global serialization. This means that two threads executing in an object should contact some external server object for serialization. Thus this solution becomes very expensive.

Another solution is to make the object single threaded. In this case we may loose some concurrency. More research is needed to evaluate the performance of these two solutions. Currently we just assume single threaded object.

Load Sharing: One advantage of our replication scheme (where all the replicas are active and process the client requests) is that the replicas can share the server load. If a client only wants to read the object (i.e., the invoked method does not change the state of the object), then the client can contact any one of the replica to get the result. However the exact mechanism of *load sharing* depends on the replica consistency algorithm. It is also possible to incorporate the application semantics/knowledge to devise an efficient load sharing strategy. Without knowing the application semantics, one simple strategy that can be used to share the server load is the *random selection scheme*. In this scheme the client will randomly choose one of the replica for servicing its read request. However this scheme can cause some consistency problem at the client side as explained below (this situation applies to our master-slave replica consistency algorithm).

Assume that client 1 is updating a replicated object O_1. The master object and the slave replica 1 is updated and the master is forwarding the update to slave replica 2. Now another client (say client 2) randomly picks slave replica 1 and reads the last updated state. The client 2 wants to read O_1 again, and this time it picks slave replica 3. But replica 3 has not yet been updated. Thus client 2 will not get the last update for this read. Usually this situation is not acceptable.

One solution for this problem is to let the client always choose only one replica for reading. But how do we assign the read replicas to the clients? One solution is to statically partition the possible client set and pre-assign the servers. But a better solution that we propose is to enhance the naming service functionality. When a client resolves the object name from NS, the NS will also allocate the read replica for this client. In this way the load distribution will be more efficient.

7 Implementation

We implemented a preliminary version of the prototype of our object replication framework. Our implementation environment is the Spring OS. Spring is a micro kernel based object oriented distributed operating system[19]. All services in Spring (user services as well as system services) are implemented as objects. All objects in Spring have well defined interfaces, and these interfaces are defined using the Interface Definition Language (IDL). An interface is a *strongly typed contract* between the implementation (object manager) and the *clients* of the object. IDL interface only defines the operations (with necessary arguments) on the object. The object developer has to implement the actual function of the object. IDL supports multiple inheritance of the interfaces.

Initially we used Spring's *subcontract* facility to implement the preliminary version of this architecture[23]. As we gained experience in the requirement for developing replicated objects, we modified the architecture to the current form. For the prototype implementation of the current architecture, we *hand coded* the object interfaces for all the objects shown in the Figure 3. We implemented the master-slave replica consistency algorithm for our architecture. The implementation of the load sharing feature is not complete at present. We are yet to obtain the performance figures for the replica consistency algorithm.

8 Conclusion

In this paper we described an architecture for the development of replicated objects in distributed systems. The architecture is modular and simple. It identifies and provides the necessary facilities for developing replicated objects. It provides a base for supporting various replica consistency algorithms. We also developed an efficient replica consistency algorithm. This algorithm uses the features of our architecture for its advantage. A preliminary prototype implementation experience was encouraging. For the future, we want to develop the language preprocessor and modify the IDL compiler such that it would automate the whole process of developing replicated objects as specified in the architecture.

References

[1] P. Alsberg and J. Day, "A principle for resilient sharing of distributed resrources," In *Proc. Of Second Intl' Conf. on Software Engg., San Francisco, CA.*, pp. 562–570, 1976.

[2] K. P. Birman et al., "Implementing fault-tolerant distributed objects," *IEEE Trans. Softw. Eng.*, vol. 6, no. 11, pp. 502–508, June 1985.

[3] P. Jalote, "Resilient objects in broadcast networks," *IEEE Trans. Softw. Eng.*, vol. 15, no. 1, pp. 68–72, January 1989.

[4] F. Schneider, "Implementing fault tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, December 1990.

[5] N. Budhiraja et al., "The primary-backup approach," In *Distributed Systems, 2ed Edition*, S. Mullender, editor, pp. 199–216, Addison-Wesley, 1993.

[6] E. C. Cooper, "Replicated distributed programs," In *ACM Symp. on Oper. Syst. Princ.*, pp. 63–78, 1985.

[7] M. Olsen, E. Oskiewicz, and J. Warne, "A model for interface groups," In *10th Symposium, Reliable Distributed Systems*, pp. 98–107, 1991.

[8] M. C. Little, *Object Replication in a Distributed System*, PhD thesis, Computer Science Dept., University of Newcastle upon Tyne, September 1991.

[9] B. Liskov et al., "Replication in the harp file system," Technical Report MIT/LCS/TM-456, Massachusetts Institute Of Technology, August 1991.

[10] G. Beedubail et al., "Fault tolerant objects in distributed systems using hot replication," In *Proc. of 15th Int'l Phoenix Conf. on Computers and Communications (IPCCC'96), Phoenix, AZ*, March 1996.

[11] T. Joseph and K. Birman, "Exploiting replication in distributed systems," In *Distributed Systems*, S. Mullender, editor, pp. 319–367, Addison-Wesley, 1988.

[12] P. Dasgupta, R. Leblanc, M. Ahamad, and U. Ramachandran, "The clouds distributed operating system," *IEEE Computer*, vol. 24, no. 11, pp. 34–44, November 1991.

[13] M. F. Kaasshoek, A. S. Tanenbaum, and K. Verstoep, "Using group communication to implement a fault-tolerant directory service," In *International Conf. Distributed Computing Systems*, pp. 130–139, 1993.

[14] J. S. Spiewak, "Replication in spring: A new subcontract," Technical Report CS-95-16, Brown University, May 1995.

[15] C. Pu, J. Noe, and A. Proudfoot, "Regeneration of replicated objects: A technique and its eden implementation," *IEEE Trans. Softw. Eng.*, vol. 14, no. 7, pp. 936–945, July 1988.

[16] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat, "Lazy replication: Exploiting the semantics of distributed services," Technical Report MIT/LCS/TR-484, Massachusetts Institute Of Technology, July 1990.

[17] T. Mann, A. Hisgen, and G. Swart, "An algorithm for data replication," Technical Report 46, DEC Systems Research Center, Palo Alto, CA, June 1989.

[18] E. N. Elnozahy and W. Zwaenepoel, "Replicated distributed processes in manetho," In *Digest of papers: The* 22^{nd} *Int. Symp. Fault-Tolerant Comp.*, pp. 18–27, 1992.

[19] J. Mitchel et al., "An overview of the spring system," In *Proceedings of of Compcon Spring 1994*, February 1994.

[20] T. J. Mowbray and R. Zahavi, *The Essential Corba: System Integration Using Distributed Objects*, John Wiley & Sons, Inc., 1995.

[21] G. Beedubail, P. Kessler, and U. Pooch, "Replicated naming service in spring," Technical Report TR95-048, Computer Science Department,Texas A&M University, December 1995.

[22] S. Radia, M. Nelson, and M. Powell, "The spring name service," Technical Report SMLI-9316, Sun Microsystems Laboratories, 1993.

[23] G. Beedubail, P. Kessler, and U. Pooch, "Object replication in spring using subcontracts," Technical Report TR95-041, Computer Science Department,Texas A&M University, September 1995.

Appendix A

This appendix describes the details of the objects (see Figure 6) shown in the Figure 3 in Section 3. Object *Mobj* provides the management functions. The *add()* is invoked by an object (say O_i) that wants to join the replica group. The parameters of *add()* specify the object reference of O_i and its requested position in glist. The *IPaddr* (IP address) is used to avoid the duplicate object references in the *glist* . The *del()* function deletes the replica on a given node from the group. The *state_xfer()* and *glist_xfer()* transfers object state and glist from the invoking object to the invoked object. It is assumed that the developer provides routines to *marshal* and *unmarshal* the object state such that it can be transmitted over the network. The *list_replicas()* routine is a utility function that shows the host machines of the current replicas. The *ping()* routine is used for testing the *liveliness* of the objects.

The *glist* and *prot_parms* are simple structures. The *update_log* is a local object at the server replica. Its methods are used to store and retrieve the invocation parameters/results. The method *chkdup()* can be used to check whether a given invocation with a given sequence number has already been executed at this object. To make the store/retrieve the parameters/results independent of the application logic, we use the *marshalled* form of the parameters and results. The developer (or the compiler) has to do this marshalling/unmarshalling while using the *update_log*.

```
interface Mobj {
      void      add (IN obj_ref, IN pos, IN IPaddr);
      void      del (IN IPaddr);
      void      state_xfer(IN state);
      void      glist_xfer(IN glist);
      glist     list_replicas();
      long      ping();
};

interface Sobj {
      // Application dependent, e.g.:
      void      set (IN key, IN val);
      val       get (IN key);
};

interface Psobj : Sobj {
      //Nothing here, inherits all methods from Sobj.
      //This will act as a proxy for replicated object
      //at the local machine.
};

interface Rsobj : Sobj, Mobj {
      //For all update methods in Sobj
      //define corresponding method
      // with prot_parms as the last argument.
      //i.e., for set (IN key, IN val) define
      void      set_r (IN key, IN val, IN prot_parms);
};
```

```
Interface update_log {
      int       chkdup(IN seqNo);
      void      get_parms (IN GUN, OUT mb);
      void      set_parms (IN GUN, IN mb);
      void      get_results (IN GUN, OUT mb);
      void      set_results (IN GUN, IN mb);
};

Struct mb {// marshal buffer
      seqNo_t       seqNo;
      int           method_no;
      sequence      value;
};

Struct glist {
      ipaddr_t      ipaddr;
      obj_ref_t     oref;
      int           status; // UP or DOWN ;
};

Struct prot_parms {
      seqNo_t       seqNo;
      int           RetCode;
      // other protocol dependent variables
      // as necessary.
};
```

Figure 6: Interfaces (objects) used in the development of replicated object.

Appendix B

This appendix describes our replica consistency protocol. Section 4 explained the working of the protocol. Here we will present the details of the protocol in pseudo code and sketch the proof of correctness. Figure 7 shows the pseudo code for the protocol. We use *one copy serializability (1SR)* as the correctness criteria[8].

Outline of Proof: We use the *master-slave* configuration. Here the clients always contact the master. Hence clients always see one copy of the object. Thus to prove that our replica control algorithm is 1SR, we should show: (1) There will be only one master in the system, and (2) Updates are not lost due to (partial) failures.

Note that, there need to be only one master in the system when any client wants to access the object. If the system is in an idle state (i.e., no client accesses the object) and if the current master dies, we need not elect (create) a new master immediately. Only when a client accesses the object the next time, will there be a master in the system. Thus statement 1 above can be modified as *there will be at most one master in the system at any time.* The following lemmas implies the correctness of the algorithm (i.e., implies 1SR). For keeping this discussion short, the lemmas are stated without proof.

Lemma 1 *At any time t there will be ≤ 1 master in the system.*

Lemma 2 *Following is the Update Semantics guaranteed by the algorithm:*

A: *If the client does not fail during the update, then all the UP servers will receive the update.*

B: *If the client fails during the update then one of the following will happen :*

1. *Update is not recorded in any of the servers.*
2. *Eventually all the UP servers will receive the Update.*

C: *All the updates are received in the same order in all the servers (Each update is uniquely determined by GUN).*

Lemma 3 *Following is the read semantics guaranteed by the algorithm.*

Any client that reads the object gets the latest state of the object consistent with its prior requests. That is, a client reads the value it had written. It may not read the latest update performed by the other clients.

Theorem 1 *The Algorithm is 1SR and is fault tolerant.*

```
invoke_remote_proc(parms)
{ // This method is executed by the proxy object.
  // This implements the client's part of the
  // replica consistency protocol

  while (TRUE) {
    send invocation to the first object
      in the glist that is alive;
    if (success) return (results);
    if (a new master reference returned)
      update_glist();
    if (all the obj_refs in glist are stale)
      get the fresh glist from the NS;
  }
}


// For adding/restarting (recovering) a replica we
// run the r_sobj load module with one parameter
// that indicates the position of this replica
// in glist. If pos = 0, then this replica takes
// over as the master.

// The server maintains the protocol parameters:
// GUN (global update number), Master_flag, glist,
// and the update_log object.

r_sobj(pos)
{ // This routine is invoked by the joining replica
  Master_flag = NO;
  glist = resolve_ns ("name");
  if (glist == empty) {// this is the first replica
    create an instance of Rsobj;
    update glist;
    bind the glist in NS;
    Master_flag = YES;
    wait_for_client_requests();
  }

  // We are here ==> object is joining the group
  oref = glist[0].oref;//the first alive object
      //or the object ref of current master
  my_oref = create an instance of Rsobj;
  oref.add(my_ref, pos, my_ipaddr);
  if (pos == 0) Master_flag = YES;
  wait_for_client_requests();
}

//NOTE: For read methods the server object
//(master or slave) simply returns the results.
```

```
server_method_r(parms, prot_parms pp)
{
  if (pp.forward_flag) {
    // this is a forwarded request ==> I am slave
    if (update_log.check_dup(pp.seqNo)) return;
    else {
      store parameters in update_log object;
      local_update(); GUN++;
        //Performs the actual application logic
      store the results in update_log object;
      return;
    }
  }
  // We are here ==> invocation is for the master.
  if(! Master_flag) {
    invoke ping() on all objects whose object_refs
      are before my object_ref in my glist
    if (ok received)
      return (master_ip, master_oref);
    else {
      // Now I have to become master
      for(all slaves) send the last update
      Master_flag = YES;
    }
  }
  // We are here ==> I am the master
  dflag = update_log.check_dup(pp.seqNo);
  if (! dflag ) {
    store parameters in update_log object;
    local_update(); GUN++;
      //Performs the actual application logic
    store the results in update_log object;
    FORWARD invocation to slaves;
    return (results);
  }
  if (dflag ) retrieve results from update_log object;
  return(results);
}

add(oref, pos, ipaddr)
{ // This method is executed by the server replica
  // This follows the MS protocol given above; This
  // replaces the application logic mentioned above.

  update_glist(oref, pos, ipaddr);
  transfer glist for all slaves;
  oref.st_xfer (my_state);
  bind the new glist in NS;
  if (pos == 0) Master_flag = NO;
}
```

Figure 7: Replica Consistency Algorithm.