Object Replication in Spring Using Subcontracts

Ganesha Beedubail[†], Udo Pooch

Department of Computer Science, Texas A&M University,
College Station, TX - 77843.

and

Peter Kessler

Sun Microsystems, Inc., 2550 Garcia Avenue,
Mountainview, CA - 94043.

Abstract

This paper presents the design of a framework for the development of replicated objects in Spring (object oriented, distributed) Operating System. For this framework, we make use of the subcontract abstraction provided by Spring. We designed two subcontracts (replicon and repleton) which can be used to develop replicated objects. We also specify the methodology for developing a replicated object server. The object replication achieved by our design is transparent to the clients (users) of the (replicated) object. The framework provides mechanism for the server developer to plug in his replica consistency algorithm. We used this design to develop a replicated naming service (RNS) for Spring. We also developed a Master-Slave protocol for replica consistency and fault tolerance. The novelty of this protocol is that, it is specified and implemented using only RPCs (synchronous communication). Lower level message passing mechanism is not exploited (which is difficult to program). The protocol is self contained. It does not assume the existence of any lower level reliable services other than simple timeout mechanism to detect failures. The paper also presents preliminary results measured from the prototype implementation.

1 Introduction

This paper describes the design and development of two *subcontracts (replicon and repleton)*, that can be used to support object replication in Spring. The concepts and design is general in that this can be used for similar purpose in any other similar platform (object oriented distributed operating system).

Object replication can be used in distributed systems for *fault tolerance* and/or *load balancing*. When an object (service) is replicated on multiple machines,the object remains available as long as one of the machine remains operational. Since replicated objects are available on multiple machines, they can share the load of client requests. This aspect becomes more attractive when the majority of the service requests are of read only (query) type.

[†]This work was done when this author was a student intern at Sun Microsystems, Inc., CA. during Summer 95.

This work was motivated by the availability of subcontract abstraction in Spring Operating system[1, 2, 3, 4, 5]. Subcontract abstraction is developed for providing a flexible base for distributed programming. We also wanted to develop a fault tolerant distributed protocol using only RPC mechanism. The desirability of such a protocol is obvious in the object oriented distributed systems where the communication between the objects is done using remote object (method) invocation. The low level message passing abstraction is not available in this case.

This paper is organized as follows. The next section briefly explains the operating environment (that is, Spring Operating System and Subcontract mechanism). Section 3 provides a brief over view of the related research work in the literature. Section 4 provides the details of the framework developed for object replication in Spring. Section 5 presents an overview of replicated naming service (RNS), that uses the concepts developed in section 4. Section 6 concludes this paper.

2 System Environment

The object replication framework we developed was intended to work on Spring Operating system using subcontract abstraction. In the following we will briefly describe the Spring OS, Subcontracts in Spring and Spring Naming Service. For a detailed descriptions of these topics refer to [1, 3, 2, 4, 5].

2.1 Overview of Spring OS

Spring is a micro kernel based object oriented distributed operating system[1]. 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 in 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.

Spring micro kernel implements the nucleus object. Nucleus manages processes, interprocess communication and virtual memory management. The nucleus supports three basic abstractions: a) *domains* (analogous to Unix processes) provide address spaces for applications to run, b) *threads* execute within domains and c) *doors* support object oriented calls between domains. A *door* describes a particular entry point to a domain. Object invocations across the network are handled by *network proxies*. Network proxies connect the nuclii of different machines transparently.Spring provides other standard system services as user level objects. These include *file system service, machine name service, TCP/UDP/IP service, tty service* etc. For details refer to [1, 2].

2.2 Spring Subcontracts

Spring provides a flexible mechanism for plugging in different kinds of object run time machinery. This mechanism is known as *subcontract*. A subcontract is a definition of a *representation* for an object and corresponding invocation protocol. An object implementation selects a particular subcontract to use for its representation. Clients are unaware of the subcontract used for the objects they possess. Subcontracts also specify how the objects are marshalled and unmarshalled when objects are used as parameters in a method invocation.

A simple, common subcontract, that is widely used in Spring is *Singleton Subcontract*. Singleton represents each object as a single door. Any call to the object is converted to an invocation through this door. Another example os subcontract is *caching subcontract* which is used by cacheable objects. For the details of subcontracts refer to [4]. We developed two subcontracts, *replicon* and *repleton* to support object replication in Spring.

2.3 Spring Naming Service

Spring provides a uniform name service. Any object can be bound to any name. This applies whether the object is local to a process, local to a machine or resident elsewhere in the network. Names have meaning only in the context in which they are used. The Name Service (NS) allows an object to be associated with a name in some context. A name is said to *denote* an object. A *context* is an object that contains a set of *name-to-object* association or *name bindings*. In Spring a *context* is any object that supports *context interface*.

Resolving a name is an operation on a context to obtain the object denoted by the name. *Binding a name* is an operation on a context to associate a name with a particular object. These operations return or take as a parameter, the object itself.

The Spring NS is implemented by *naming_context* interface/object (simply called, context object or context). The contexts are first class objects, they can be accessed and manipulated directly. By binding contexts in another contexts, we can create a *naming graph*. An inner node in the naming graph can be referred by a *compound name*. In Spring, the NS is implemented by one or more object managers called name servers. Name servers implement the context objects. For more details about Spring NS refer to [5, 6].

2.4 Objectives of this Project

The following discussion highlights the motivations for carrying out this project.

- We wanted to exercise the features of *subcontract* concept, to show that one can develop different subcontracts as the application requirements for various semantics (may be to provide variations of the base application).

- As one can see, Singleton subcontracts are general for a wide variety of applications if the client server communication is one to one.

  But when client server communication is not one-to-one, is it possible to develop a general subcontract that works for a class of applications? If the answer to this is yes, then design such a subcontract. Otherwise identify the limitations.

- Design and implement a subcontract for a specific application in a class of applications and show how it can be done. We can also give a general guideline as how to write/modify the subcontract for the other applications of the same class. The application we selected is replicating objects in Spring (because no such subcontract exists now). The replicated object system so developed should be transparent to the client application. But it may not be so to the server developer. However, delegate the work to the subcontract as much as possible.

- Explore the possibility of designing fault tolerant (and consistency control) algorithms using only RPC facility. Since currently RPC is the high level communication primitive available to the developer of a object oriented application system. Even though low level communication primitives are available (like TCP/IP or UDP/IP) using them effectively in practice requires more time and effort. Also in the case of Spring, these message passing primitives are exclusively hidden from the user (it provides transparent network proxy mechanism for remote object communication. Bypassing this proxy mechanism considered not a good practice). So designing a protocol that uses only RPC is a necessity in Spring.

3 Related Work

In this section we will briefly review the related research work. As we said earlier, our work can be considered as a (special) case of object replication, we will consider research work on object (process/service) replication.

Replicated (fault tolerant) objects/services are generally structured in one of the following two ways (or the 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*[7]. 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[1]. If the primary fails, then one of the backups becomes the primary. This approach is known as the *primary-backup* or the *primary copy*[8]. All the research work on object replication that we are aware of are the variations or combinations of the above two basic strategies[2].

Alsberg and Day [8] proposed one of the earliest primary-backup protocol. The paper presents the error detection and recovery schemes for *two-host resiliency* (tolerating one failure) and the extension of this concept n hosts. In [9] Budhiraja et. al. present some theoretical aspects of primary backup approach. Their system assumes closely synchronized clocks in the system. They

[1]The algorithms vary in how the request is forwarded to the backups and when the primary responds to the client.
[2]The voting scheme or variations of the voting schemes can be implemented as variations of primary-copy scheme.

classify the primary-backup algorithms as *blocking* and *non-blocking* and provide lower bounds (under various failure conditions) on the *degree of replication, blocking time* and *failover time* for any primary-backup algorithm. They also provide protocols to prove that their lower bounds are tight. The Harp File System also uses a primary-backup protocol[10]. It is assumed that the system clocks are closely synchronized and all the machines are equipped with a UPS (uninterrupted power supply). 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 [11] Birman et. al. describe a technique to implement *k-resilient* distributed object. They use *coordinator-cohort* scheme, a variation of primary-backup scheme. The coordinator services the client requests and periodically checkpoints its state to the cohorts. The coordinator also forwards the result of any *external action* to the cohorts (called *retained results*). Thus the cohorts will have all the information to take over as a coordinator if the original coordinator fails. In [12] Beedubail et. al. extend the primary backup protocol for *nested object invocation* where to service a client request and object can invoke the services of other objects. That scheme assumes the existence of a reliable multicast and group management service.

The *replicated distributed programs* described by Cooper[13] is an example of active replication. Here, all the replicas (called the members of a *troupe*) execute the client calls independently. A higher level application protocol should make sure that all the replicas get their client calls in the same order. In [14] Birman et.al., describe, how the *totally ordered, reliable broadcasts* can be used to maintain *replicated data* (active replication).

Object Replication in Arjuna[15] uses active replication[3]. The replication algorithm uses the support of *naming and binding* service and the *atomic action service* provided by Arjuna system. A higher level group locking (issued by the application program) protocol avoids the need of reliable ordered broadcast. In [16] kaashoek et.al., describe the *fault tolerant directory service* implemented in Amoeba distributed operating system. They use the *closed group communication* service supported in Amoeba to implement the directory service. Our work provides a mechanism for implementing object replication. The server developer can use any of the above algorithms. We also illustrate the feasibility of our framework by implementing Replicated Naming Service using Master-Slave (Primary-Backup) protocol.

4 Framework for Object Replication in Spring

The task of replicating objects in Spring should address following issues:

Design of Object Representation: We need to come up with a representation mechanism for replicated objects.

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

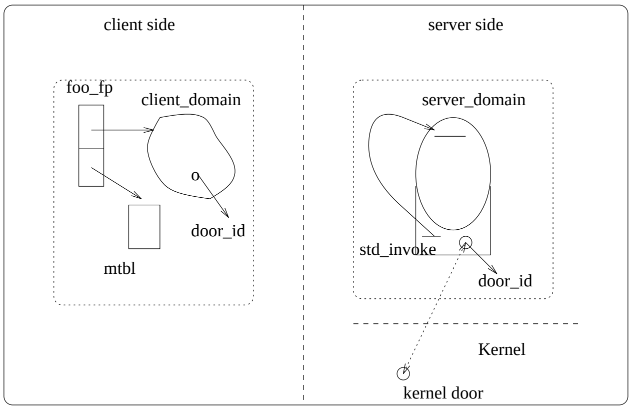

Figure 1: Representation of a Singleton Object

Design of Control Mechanism: We need to design mechanisms to start a replicated service, add the members to the replicated service or remove the members from replicated service. We call this the *Server Development Framework*.

Invocation Mechanism: This should describe how one can access the replicated object.

Replica Consistency Protocol: We need to develop protocols to keep the states of the replicated objects in consistence with each other. More over, the protocol should be fault tolerant.

In the following, we will explain in detail, how the above issues are addresses in this particular paper.

4.1 Object Representation

Before explaining the representation of replicated object, we need to understand the representation of singleton object (i.e., non replicated object). Figure 1 shows the representation of singleton object.

Although, at the higher level (application programming level) one can consider that the client has the object, at the object implementation level, we need to separate the client side and the server side representation of the object. Thus the figure shows them separately. The client side representation of the object has a *fat pointer*[4] which encapsulates the actual pointer to the object. For a singleton object it is just a *door id*. The fat pointer also has a pointer to the method table (which contains the pointers to methods of the object). At the server side, the object representation contains a door id (represents this object) and a pointer to the primary (dispatcher) function that is invoked when a call/request arrives for this object through the door.

[4] We can consider this just as a C++ pointer, we can use it for de-reference in the usual way. But it more than just a language level pointer (the object reference is encapsulated inside it), thus it is called a fat pointer.

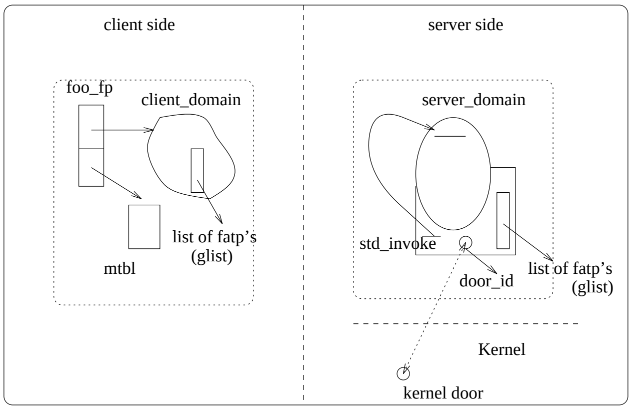

Figure 2: Representation of a Replicon Object

For the representation of replicated object, we extrapolate the ideas of singleton object representation. Figure 2 shows the representation of a replicated object. It contains a list of fat pointers at client and server side (in the case of singleton object, we just had a single door here). Note that the representation also has some more housekeeping details, which will be explained later in this paper in connection with the replicated naming service.

Some comments regarding the choice of this representation is relevant here. First, we chose a list of fat pointers rather than a list of doors for the representation. This choice lets us replicate any general type of object, not necessarily a (replication of a) singleton object. It lets us replicate, say a cached object or a replicated object (well, these will not have much use in practice, but just to illustrate the point). Secondly, it is obvious that the client should have a list of pointers to all the replicas of the replicated object. Because the client may need to contact one or more of them in the case of replica failures. But it is not very obvious that why the replicas have a list of pointers to the other replicas. It is basically provided for supporting the implementation of replica consistency protocols. We believe that most of these protocols need some kind of support for group management.

4.2 Server Development Framework

As explained in section 2, server development for replicated object is not transparent to the user. The user should develop his/her server in a pre specified manner so that the whole set up works as described. By server development, we mean the logic, which starts up the replicated object and exports (binds) it to the name space (name service). It also involves making (forming) client side representation (making fat pointers), filling in the necessary housekeeping details in the object representation, providing services for adding/deleting replicas to/from the currently running replicated service. This does not involve any restriction/changes to the actual application code (object implementation). The implementation of the application code may be the same for the replicated object and singleton object. However, the application developer has to implement the logic for

```
interface sobj {
    long  get (copy long key)
    void  set (copy long key, copy long val)
}


interface sobj_repl : sobj {
}


interface sobj_mgr : sobj {
    //Some methods that may be necessary to implement
    //consistency control protocol
}


interface sobj_register {
    void  register_sobj (copy sobj_mgr smgr)
}
```

Figure 3: Interfaces involved in Object Replication in Spring

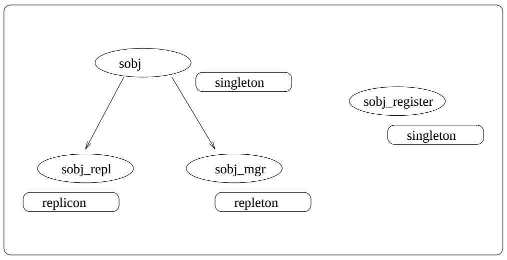

Figure 4: Subcontracts involved in Object Replication in Spring

replica state consistency. This may very well depend on the application at hand. Thus one may want to change the implementation of the application (compared to the singleton implementation) such that it becomes easier to implement the consistency protocol.

Developing a replicated object service in Spring involves the following:

- Define the object interface (using Spring Interface Definition Language, IDL) and implement the object. This is just like the implementation of singleton object. For the current illustration purpose, let this object be called *sobj (some object).*
- Define interfaces, *sobj_repl* and *sobj_mgr*. Both these interfaces inherit from *sobj* interface. Note that the names are just descriptive. They need not be named exactly like this. The *sobj_repl* interface should use *replicon* subcontract and *sobj_mgr* should use *repleton* subcontract. In Spring development environment subcontracts are specified in environment specification files (*.env files*).
- Define another interface, named *sobj_register* (or any other name, implementing the same function). This object/interface is used for making/constructing the list of objects (list of fat pointers or group list) in the replica set.

Note that the *sobj_repl* and *sobj_mgr* should implement all the the methods of *sobj*. Currently Spring only supports *interface inheritance*. It does not support *implementation inheritance*. How ever, this is not a disadvantage here, since the logic of the same methods in all the three implementations is going to be very much different. In this framework, the *sobj_repl* interface is used just to make the client side representation (and construct group list) for the replicated object. Thus we need a *dummy* implementation at the server side (since the server side of this object is never invoked for the application methods). The methods of *sobj_mgr* may implement the consistency control algorithm. The implementation of these methods should finally invoke the methods of *sobj* to actually service the client request.

Figure 3 and figure 4 shows the interfaces and the subcontracts involved in this framework. For the purpose of illustration we just defined two methods (operations) in the *sobj* interface.

It is interesting to ask the question *why we need two subcontracts here*? To answer this question, one needs to have some knowledge of the actual working of the subcontract mechanism in Spring.

Normally, when a user develops a Spring application, he first writes IDL specifications for the server interface (in xxx.idl file). The subcontract for the interface is specified in the environment file (xxx.env). If no subcontract is specified, it defaults to singleton subcontract. Then the xxx.idl file is compiled by IDL compiler, which generates client and server side stubs. The subcontract representation is embedded in the stubs generated. Thus the same subcontract is used for server and client side stubs (of course, the server side representation of the subcontract is used for server stubs and client side representation of the subcontract is used for the client stubs, see figure 1).

In our replication framework, the client should able to invoke replicas transparently (done by *replicon subcontract*). The servers should able to invoke other servers as needed by the consistency protocol. This implies that, the invocation from the client side is standard (say for a given consistency protocol). But when the server invokes its peer (other servers) the invocation is need based and depends on the current situation. Thus we need a different invocation mechanism here. Hence we introduced *repleton subcontract* for this invocation. The client side representation and invocation for *repleton* is similar to the singleton subcontract counterparts. But the server side representation and the server response for invocations are different from *singleton* subcontract. The server side of *repleton* subcontract implements part of the replica consistency control protocol and it also stores the protocol variables. The client side of *replicon* subcontract implements the client part of the replica consistency protocol.

Figure 5 shows the end effect of we want to achieve in this framework. As shown in the figure, we need multiple copies of *sobj_mgr* running on different machines (we assume that *sobj_mgr* encapsulates the *sobj* object). The *sobj_mgr* should have a list of (pointers to) all the other *sobj_mgr*'s in the replica group. The client should have a list of pointers to all the *sobj_mgrs* (replicas). In the following we will explain how this can be achieved.

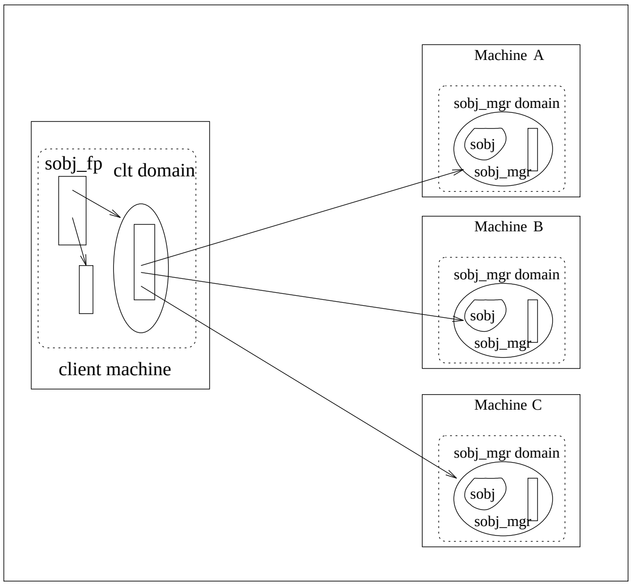

Figure 5: An overall view of the Replicated Object [The big Picture]

Figure 6 shows the high level logic for developing a control mechanism for replicated object service (i.e., to achieve the configuration shown in figure 5). Here we will assume the existence of a fault tolerant distributed naming service. We will later explain (in section 5) how this service is being implemented.

The main program creates a register object (*sobj_register*) and binds (exports) it into the name space (with name say, "services/sob_register"). Next, it creates child domains (child processes) on different machines. These child domains run *sobj_mgr* server program. The *sobj_mgr* server program creates *sobj_mgr_impl* object (which is basically the implementation of *sobj_mgr* interface methods. This also encapsulates the sobj object either with a reference to the *sobj* object or with the direct implementation of the *sobj*). It then creates a client side representation for itself (by creating a fat pointer for itself) and registers itself in *sobj_register* (passing its fat pointer to the *sobj_register* object by invoking register_sobj() method). Then it simply waits for the client request (becomes server).

The *sobj_register* object basically performs two functions: a) It constructs the group list (list containing the references to the replicas), and b) It sends the group list to all the replicas and makes the list available to the clients by binding/exporting the list (in some encapsulated form) to the name space. A more detailed description of *sobj_register* follows.

The *sobj_register object* waits until all the register requests arrive. Note that the initial count of replicas in the system can be given as a command line argument for this program. Here we also

```
main()
{
    create register object
        /* and bind/export it in the name space */
    create replicas
        /* create child domains which run sobj_mgr_srvr */
}

sobj_mgr_srvr()
{
    create sobj_mgr_impl
    register in register_obj
    become server
}

register_object()
{
    wait until all register requests received
    prepare a list of fat pointers
    send the fatp list to all sobj_mgrs
    create sobj_repl_impl
    set fatps /* in subcontract representation */
    make client side representation /* make sobj_repl fatp */
    bind in name space /* as sobj type */
}
```

Figure 6: High level logic for setting up Replicated Objects in Spring

assume that a human operator initiates this starting up process. The operator can restart the whole process if there is any failure during this registration (replicated service startup) process. Once all the register requests are received, the register object makes the list of fat pointers it received (references to all the *sobj_mgr*). This list is then sent to all the *sobj_mgr* objects. Sobj_mgr objects then update their local group lists. At this point, the *sobj_mgr* objects are ready for servicing the client/application requests).

The *register object* then creates an object of type sobj_repl_impl (the implementation of sobj_repl interface). The group list (the glist, which is in the representation of replicon/replicated object) is filled by the fat pointer list. The server part of the replicon subcontract provides a method for setting this field in its representation. The register object then makes a client side representation (fat pointer to *sobj_repl_impl* object) of this object and binds/exports it to the name service.

Note that while exporting the replicated object reference (fat pointer) to the name space, the type of the object is specified as *sobj type*. Thus the client here just sees an object of type *sobj* instead of type *sobj_repl* (thus it sees a singleton object in the name space, even though the object is replicated). At this point the set up for replicated object is complete. The clients can lookup to name service for sobj object (as if it were a non replicated object) and can get a reference (fat pointer) to the object. Then it can make invocations as usual. The object service is available as long as one of the *repl_mgr* objects is available.

4.3 Invocation Mechanism

While stating the goal of this project, we said that the replication should be transparent to the client developer. Thus the client developer need not do anything special (beyond that is necessary for the singleton object client) for the replicated object invocation. How ever, the replicon subcontract should implement some mechanism to invoke replicated object. This mechanism should be application independent, that is, it should be same for all server objects. Thus the simple invocation mechanism we provide through the replicon subcontract is : *serially invoke the replicas until it gets a (correct) response.* Here, *correct* means that, it should get a response (an answer) from the sobj_mgr (server) but not a *proxy_failure* or *bad_target* return code. In Spring, when the target machine is down or network is broken, you get a return code *proxy_failure* (basically detected by timeout mechanism). If the target domain is down then you get a return code *bad_target* for the invocation.

4.4 Replica Consistency Protocol

In the current framework for Object Replication in Spring we leave the implementation of Replica Consistency Protocol to the application developer. We believe that the Consistency required for different objects my vary widely in practice and the application developer is in a better position to specify what is required for the specific object. Providing a general protocol for every application objects may be overly pessimistic. In general, the protocol should handle failures of other replicas while maintaining the mutual consistent state. We provide one support for this in terms of group list. Since each of the replicas (sobj_mgr objects) has list of all other replicas, it is possible to implement some kind of fault tolerant protocol for replica consistency. The reader can refer to [13, 7, 12, 17, 18] for the discussions of the various approaches used for solving this problem.

4.5 Adding/Deleting Replicas

In practice, it is necessary to add new replicas to the existing replica set to ensure a high availability level (or it may be the case that, a machine hosting a sobj_mgr server goes down and come up again). In our framework, adding replicas can be done in a straight forward manner. Adding a replica can be considered as an update operation on sobj_mgr object state (in fact it is really an update operation, since the group list has to be updated). As a side effect to this update, we also need a *state transfer* operation, in which the state of a replica in the current group should be transferred to the newly added replica. Thus we can use the protocol used for replica consistency for adding a new replica to the group. Deleting a replica can be handled similarly. It is also necessary that, the newly joined replica, update the client side representation bound in the name space. Otherwise after some activity in the system, we may be left with a stale object representation for the replicated object.

The rest of the paper describes Replicated Naming Service (RNS) in Spring. RNS is developed using the framework developed above. It also illustrates how one can implement a replica consistency protocol in the above framework.

5 Replicated Naming Service (RNS) in Spring

We decided to develop Replicated Naming Service (RNS) mainly for two reasons. First, we wanted to develop a real life (practical) application using the framework developed above. The Spring team at Sun was interested in a Fault Tolerant Village Name Service (for the details of Spring Naming Service and Village Name Service refer to [5, 6]). Simply stated, Village Name Service is the shared/distributed name service available for a group of Spring Machines in a LAN. Another reason is that, in our object replication scheme, we assumed the existence of a fault tolerant naming service which needed to be realized some how.

It is necessary to modify the above scheme slightly to implement replicated name service. This is because the RNS can not use a fault tolerant naming service (simply because it is non-existing). For implementing RNS some changes were made in the replicon and repleton subcontracts. The subcontracts now implement part of the replica consistency algorithm (we use *master-slave configuration* for replica consistency). In the following, we will describe the details of RNS and the Replica Consistency Protocol.

A detailed description of Replicated Naming Service and the complete fault tolerant replica consistency algorithm will be given in a separate paper.

Note that in the following discussion, the names used do not have any special significance. Any name with similar intention can be used (although two of the names used has special significance in Spring, one need not concerned about them to understand the RNS scheme).

5.1 Brief Overview of RNS

We assume a LAN which has M machines running Spring OS. We designate N of these M machines ($N \leq M$) as name server machines (RNS machines). Our replica consistency algorithm requires the identity of all the machines that host the replicated name server. Thus we list all these N machines (the IP addresses of the machines) in a file (called RNSfile). This file is statically available to all the machines in the network. The listing order of all the machines is the same in all the files.

When a machine comes up, it checks whether it a RNS machine. If so, it starts RNS server program (we call it *rep_ns*[5]). It then starts RNSmgr program (we call it *rmgr*). The RNSmgr constructs the client side representation for replicated naming service. Note that all (Spring) machines in the network are the clients for RNS. Thus the RNSmgr program should run in all the machines when they come up. Both of these programs will be running forever. Note that the *rep_ns* program is equivalent to *sobj_mgr* and *rmgr* is equivalent to *sobj_repl* described in the previous section.

[5]The names are somewhat confusing. But we retained these names, because these names can be found in the source code of the prototype implementation.

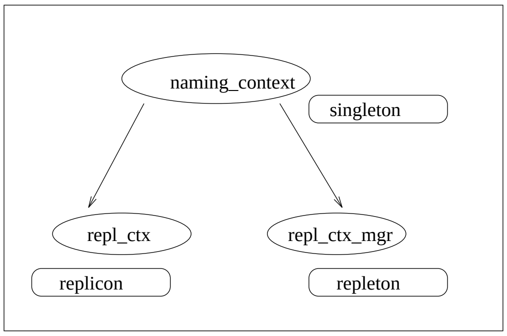

Figure 7: Interfaces and Subcontracts involved in the RNS

When *rmgr* is started, it forms a representation for the RNS servers. This representation basically is a list of fat pointers. This list is then encapsulated in the representation of a new context (contexts are roughly analogous to Unix directories). This new context is bound in the local machine name service (all spring machines have a local machine name service) as "*machine/repl_ctx*". All application objects bound to this context are automatically bound in all the replicated name servers.

5.2 Detailed Description of RNS

In this subsection we will elaborate the logic of RNS server (rep_ns) and RNSmgr (rmgr) programs. The logic of *rmgr* is comparatively simple. The logic of *rep_ns* is more involved. It has to implement the replica consistency protocol as well as replica join protocol (when an RNS machine comes up, the replicated name server started in that machine should join the group of name servers currently working). Figure 7 shows the interfaces and subcontracts involved in the RNS.

The naming_context interface currently implements all the functions required by a naming service (refer [5]). The *repl_ctx* interface (object) is used by rmgr to construct the client side representation (the list of fat pointers to the RNS servers). The *repl_ctx_mgr* interface is used by RNS server (rep_ns) to implement actual replicated naming service, consistency control protocol and the start up (join) protocol for RNS (we use *Master-Slave* configuration for replica consistency protocol).

The *repl_ctx* interface uses *replicon* subcontract for its representation. The *repl_ctx_mgr* interface uses *repleton* subcontract for its representation. The details of these representations will be given shortly.

Now we will describe how the replicated object (replicated NS) is set up (as shown in figure 5, the big picture). When RNS server (rep_ns) starts up, it creates a new naming context object in its local state. This is to implement the actual service of the naming context like *bind* and *resolve*. In the current implementation we used pre-existing services to implement this function without any modification. Note that in this case making naming service fault tolerant (or replicating naming service) simply involves the plugging of replication protocol to the existing implementation without changing the original implementation. After that, it constructs an object reference (fat pointer) for itself (for the implementation of repl_ctx_mgr object) and binds this object reference in the local machine name service as "*machine/export/services/repl_ctx[n]*". The "n" at the end of name specifies the replica_id of this particular replica. The replica_id of a replica is the position of this machine in the list of machines found in the RNSfile. Then RNS server waits for the client requests.

When the RNSmgr (rmgr) comes up (in RNS machines, it will start up after the RNS server) it will get the object references (fat pointers) for all the RNS servers. It gets these fat pointers by resolving to the following names in the local machine name space.

$$machine/services/proxy\_context/dd.dd.dd.dd/repl\_ctx[n]$$

where *dd.dd.dd.dd* represents the IP address of the machine found in RNSfile and "n" is the replica_id. In Spring Operating System, when one tries to resolve a name like above, it actually resolves the name (object) bound in the context *machine/export/services* in the machine whose IP address is *dd.dd.dd.dd*. This is transparently achieved by *proxy_context* service (with the help of network proxy service). Recall that, RNS server bound its reference exactly in the same context in its local machine name space. Thus every client machine has a way of obtaining the reference to all the RNS servers in the LAN. Note that in the case of object replication framework described in section 2 (which uses a distributed name service), the replicas register themselves in a register object. In the RNS case, we can to do this because there is not a place where we can bind/export the register object, that is known to all the RNS servers.

After the RNSmgr gets the references to all the servers it creates a repl_ctx object, fills its (subcontract) representation with the fat pointer list and IP address list (the server side glist, as shown in figure 2 is updated). Then it makes a fat pointer for this repl_ctx object and binds it in the local machine name space (as *machine/repl_ctx*). The type of this object is specified as naming_context type. Note that the true type of this object is repl_ctx type (which is a sub type of naming_context type). Thus the end user (client) can consider this as an instance of naming_context type and perform all the operations (method invocations) provided by the naming_context interface in this context (machine/repl_ctx). All these operations automatically gets replicated (i.e., a bind in this context actually does bind in all the RNS servers) because in actuality, *machine/repl_ctx* points to multiple repl_ctx_mgrs (RNS servers).

The RNSmgr periodically performs the whole of the above process. This will minimize the possibility of machine/repl_ctx becoming stale. This context becomes stale when all of the original RNS servers go down and come up one or more times after the RNSmgr bound this context. This is because, when a server dies, the fat pointer it handed out earlier becomes invalid.

Figure 8 shows the IDL definition for the interfaces used in the RNS. It just shows some of the methods/operations in the RNS. It does not strictly conform to the syntax of Spring IDL and the

```
interface naming_context {
      void                  bind(name, named_object, ...);
      void                  rebind(name, named_object, ...);
      named_object          resolve(name, ...);
      ..                    ..
};

interface repl_ctx : naming_context {
};

interface repl_ctx_mgr : naming_context {
      void                  bind_forward(name, named_object, seqNo...);
      void                  rebind_forward(name, named_object, seqNo ...);
      ..                    ..
      // join_group method is invoked by some server, which wants
      // to join the group. As a side effect of join, the state
      // of the (current) master is transferred to the joining member.
      // [using naming_lib::copy_naming_tree(src_fp, dst_fp)]
      //
      long                  join_group(me, my_id, my_ctx_fp);
      long                  join_forward(me, my_id, seqNo);
      long                  are_you_ok();
};
```

Figure 8: IDL file for RNS

argument list is not complete. It is just for the purpose of illustration.

All the operations (methods) defined in *repl_ctx_mgr* interface (the xxx_forward and join_group) operations are used for implementing replica consistency protocol.

5.3 Details of Object Representation

Here we will briefly describe the actual representation for the *replicon* and *repleton* subcontracts that are used to represent the objects in RNS. The replicon subcontract is basically used to hold the client side representation. The server side representation of replicon subcontract is used temporarily (it is basically used to conform to the programming/application development style in Spring). The repleton subcontract is used to represent the server side of the replicated object (the RNS server). Figure 9 shows the relevant definition for replicon and repleton subcontracts. It only shows the client side definition for replicon and server side definition for repleton subcontracts.

We will briefly explain the role of each variable defined in the subcontracts. The *obj* field in replicon encapsulates the pointers to replicon subcontract methods (not relevant for our discussion here). Door identifier (*di*) in replicon is retained for compatibility purpose. It is not used in RNS. The sequence_number (*seqNo*) is provided for supporting consistency protocol (to assign sequence numbers for the RPC invocations). *Total_reps* represents the total replicas (total RNS servers). *Glist* contains the list of fat pointers to the RNS servers. *Total_upd_methods* stores the number of update methods in the interface and *update_methods* is an array of method numbers for update methods. We specify the update methods so that only requests for these methods should execute the consistency protocol. The read requests need not execute the protocol. The rest of the fields at the server side has the obvious meanings.

```
class replicon
  {
    any_obj              obj; /* pointer to this stored in fat pointer */
    door_identifier      di;
    sequence_no          seq_no;
    int                  total_reps;
    gl_type              glist[MAX_REPS];
    int                  total_upd_methods;
    int                  update_methods[MAX_UPD_METHODS];
  }

class repleton_srvr
  {
    door_identifier      di;
    probe_method         probe_are_you_ok;//this is set by the object implementor
    sequence_no          seq_no;
    int        total_reps;
    gl_type    *g_list; // the list points to the group of replicated servers
    int        total_upd_methods;
    int        update_methods[MAX_UPD_METHODS];
    int        replica_id; // the replica id for this server
    int        master_flag; // YES(1) ==> master, NO(0) ==> slave
    int        ready_flag; // YES(1) ==> ready to server, NO(0) otherwise
    mutex      repleton_mutex; // for serialization, all update methods
               // should lock this mutex
  }

typedef     struct {
            int               ip_addr;
            any_fatp          rep_fp;
            int               status; /* UP or DOWN */
            }gl_type;

gl_type     g_list[GLIST_SIZE];
```

Figure 9: Replicon and Repleton representation

5.4 Fault Tolerant Replica Consistency Protocol

We use *Master-Slave*[12] configuration for replica consistency control and fault tolerance[6]. The novelty of this algorithm is that:

- It is self contained. It does not use any other services (other than the basic "*time-out*" mechanism to detect failures.
- It is specified and implemented using only RPCs (synchronous communication). To our best knowledge, this is the first protocol which does so.

The following are the assertions made regarding the algorithm:

- Everyone in the system (all of the RNS servers and RNS clients) will have a group list (*glist*). The type(format) of each entry in this list is shown in figure 9.
- The order of the IP address list in the above list (glist) is fixed (and the same) in all the machines involved (read from RNSfile).
- *rep_fp* (fat pointer to the replica) and *status* changes as the machine go down and come up.

[6]Here we describe the algorithm as it is currently implemented. This is not complete in all respects. A complete algorithm will be presented separately.

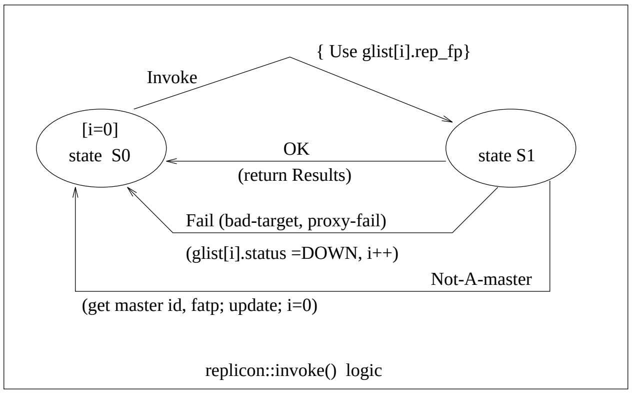

Figure 10: Replica consistency Protocol executed by the client

- 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.

  Note that maintaining this property requires that, if the machine in which replica zero (0) resides is UP, then it is always the master. When machine "i" comes up, if machine "j" is the master and "i < j", then "i" assumes the mastership.

5.4.1 Client Side Algorithm

Figure 10 shows the part of replica consistency algorithm that is executed by the client (the client invocation, implemented by replicon subcontract in *replicon::invoke()* methods).

The client sequentially invokes the replicas in the group list (glist). It starts at index zero (i=0). For an invocation, it can receive three kinds of responses as described below.

- It receives an OK response. Then the invocation is done. The result it returned to the client application.

- It receives a failure indication (a failure return code). It can be either a *proxy_failure* (this implies the target machine crash or network failure - detected by the timeout) or a *bad_target* return code (bad_target implies that the target domain is invalid - it might have crashed). In either case, the client simply invokes the next UP replica in its list.

- It can receive a response which says that the target invoked is not he current master. This response also returns the reference (fat pointer) to the current master and the replica_id of the current master. Note that this may not be the true master, but that is what the current target believes. The glist is appropriately updated and the index is set to zero. The whole process starts from the initial state.

```
repleton_srvr::_std_invoke(srvr, marshal_buffer, method)
{
    if (!is_update_method(method))
        { call_server_method(); return}

    // if we come here, it is an update method
    if (is_master())
        { call_server_method(); return}
    else {
        send are-u-ok() message to servers
             whose ID is less than mine;
        if ( YES received ) return (I-am-not-master, Info)
        else set-master-flag(YES) // I am master now
             { call_server_method(); return}
    }
}

server_method()  //if update method
{
    for (i=my_replica_id+1; i<MAX_REPS; i++){
        forward_operation();
    }
    perform local operation
}
```

Figure 11: Server Side Replica Consistency Protocol

5.4.2 Server Side Algorithm

Figure 11 shows the part of replica consistency algorithm that is executed by the server. Part of this is implemented by the server side subcontract by repleton_srvr in repleton_srvr::_std_invoke() (this is the point where the client request first arrives). The actual server method implements the rest of the protocol.

5.4.3 Replica Join Protocol (Server Start up Algorithm)

Figure 12 shows the algorithm executed by the RNS server when it first starts up (i.e., when the machine comes up). The same algorithm is executed when the server machine restarts after a crash.

5.5 Proof of Correctness

A detailed specification of this algorithm and the proof of correctness will appear in a separate paper.

5.6 Performance Results

In this section we will present the measured results from the prototype implementation. Figure 13 shows the measured results for the prototype implementation of the Replicated Naming Service. All measurements are taken on Sparc 10 machines Running Spring OS. The client and server processes are run on different machines. In the failure free case (normal operation), the response time for an update (rebind) request grows almost linearly with the number of replicas (as one can expect). The read request (resolve) takes the same time irrespective of the number of replicas (because read

```
main()
{
    get_my_ip_addr();
    total_servers = fill_glist(); // from rnsfile
    replica_id = get_my_replica_id(); // my position in rnsfile
    create_and_bind_new_context();// repl_ctx_mgr_impl()
    set update methods;
    set replica_id, probe_method() /* are-u-ok method */
    join_the_group()
}

join_the_group()
{
    I_am_ok = NO;
    while (I_am_ok == NO ) {
        get_the_other_servers_refs();
        idx = index_of_the_first_up_server();
        if (idx == my_replica_id)
            ret_code = become_master(idx);
        else
            ret_code = become_slave(idx);
        if (ret_code == OK)
            I_am_ok = YES;
    }
}
```

Figure 12: Replica Join (Server Start up) Protocol

For Rebind and Resolve, 1000 iterations were performed.
Client is executed in a separate machine.

name servers	Rebind	Resolve
1	15 ms	4.5 ms
2	23 ms	4.5 ms
3	30 ms	4.5 ms
4	40 ms	4.5 ms
5	48 ms	4.5 ms

Local Rebind/Resolve without RNS :

	Rebind	Resolve
	1.5 ms	1.1 ms

Local Rebind/Resolve with RNS (operation on machine/repl_ctx) :

	Rebind	Resolve
	10 ms	2.2 ms

Response time for a Rebind When there is a Failure :
(Failure is simulated by pulling the network cable of a RNS machine)

Rebind 52 seconds (1 machine out of n/w = 1 machine failed)
Rebind 78 seconds (2 machine out of n/w = 2 machine failed)

Figure 13: Performance Results

request is not forwarded to other replicas[7]). When there is only one replica, the resolve time is 4.5 ms and the rebind time is 15 ms (the client and server in different machines). This should be compared to the case when the client and servers are in the same machine. The values are 2.2 ms and 10 ms respectively. The difference has to be attributed to the (variable) network delay. We can also compare this results to the resolve and bind operations on local machine NS. The values are 1.1 ms and 1.5 ms. The implementation of RNS protocol logic may be responsible for the high values found in the local resolve/rebind operation of RNS.

We also measured the response time for rebind when there is a failure (machine failure is simulated by pulling the network cable of that machine). The rebind time was 52 ms when the first master was removed from the network and it was 78 ms when the first and second machine (in the RNSfile) were removed from the network. These timings depend on the *timeout* logic of Spring Proxy protocol. Also this value is taken for only one rebind (we can not execute this in a loop, because when the client detects the failure for the first time, it updates its glist and that machine is not tried again).

6 Conclusion

In this paper we presented the design of a framework for developing replicated objects in distributed object oriented system. We did a pilot implementation of the designed (general) framework on Spring Operating system and tested its feasibility. We found that it is quite satisfactory.

We also implemented a replicated naming service (RNS) using the above design. This implementation prototype is currently working. Specially, it meets our design goals. Now an application can *bind* and *unbind* an application level object into *"machine/repl_ctx"* and it automatically gets bound/unbound in all of the (UP) RNS servers. There is no need for any changes to be done for the application programs to use the RNS (not even recompiling or re-linking). We can use the pre-existing commands on machine/repl_ctx. The commands like *sls* (list the contents of this context/directory), *mkdir* (make a new context/directory), *bind, unbind* (binding and un binding of primitive objects at command level) works on machine/repl_ctx. The compound name bind/resolve also works transparently.

We support all read operations of naming_context interface. The update operations currently supported are *bind, unbind, rebind* and *bind_new_context*. However, since the implementation of protocol is not complete yet (we have not yet made use of the sequence numbers for detecting the duplicated RPC requests), it will not tolerate some failure cases (failure of the master during the forward operation). But we will include this feature shortly.

The current implementation does not include support for access control related operations in the naming_context interface. The protocol also will not tolerate network partition failures. For tolerating network partition failures, we need to enhance the consistency protocol to include some

[7] In the implementation, the read requests are randomly sent to any UP server in the client's glist.

kind of majority voting scheme. We also need to enhance the replica consistency protocol so that the RNS works when the name space is implemented by multiple name server domains. This is somewhat harder problem. We may be able to adapt the solution given in [12] for this situation.

Another feature that needs some improvement is with respect to the stale fat pointers (stale object references). If a client gets fat pointer (replicated object reference) and keeps it for a long duration, then the fat pointer may become stale (the original replicas may have died and new ones might have joined the group). One solution we have for this problem is to include the name of the object (the name with which it is bound to the Naming Service) in the representation of the object. Then during the invocation if all the fat pointers in the glist are invalid (stale), we can look up the name service and can get the up to date reference to the replicated object.

Acknowledgments: The authors would like to thank Graham Hamilton, Stuart Ritchie, Sanjay Radia, Anil Shivalingaih and Patrick Fu of Sun Microsystems for their participation in the design discussion and helpful comments and suggestions. The authors acknowledge the contribution of the members of the DSL (distributed systems laboratory) at Texas A&M University for the development of the fault tolerant replica consistency algorithm used in the replicated naming service.

References

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

[2] S. Radia et al., "The spring object model," In *Conference on Object-Oriented Technologies(COOTS)*, February 1995.

[3] G. Hamilton and P. Kougiouris, "The spring nucleus: A microkernel for objects," In *Proc. of 1993 Summer Usenix Conference*, June 1993.

[4] G. Hamilton, M. Powell, and J. Mitchell, "Subcontract: A flexible base for distributed programming," In *Proc. of 14th Symposium on Operating Systems Principles*, December 1993.

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

[6] S. Radia, "The spring naming policy," In *1st Intl. Workshop on Services in Distributed and Networked Environments*, 1994.

[7] 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.

[8] 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.

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

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

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

[12] G. Beedubail et al., "Fault tolerant objects in distributed systems using hot replication," Technical Report TR95-23, Computer Science Department,Texas A&M University, April 1995.

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

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

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

[16] 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*, 1993.

[17] K. Yap, P. Jalote, and S. Tripati, "Fault tolerant remote procedure call," In *International Conf. Distributed Computing Systems*, pp. 48–54, 1988.

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

Appendix

Future Work in RNS

This Appendix highlights the the work that need to be done for RNS (Replicated Naming Service) to make it a complete prototype.

1. We need to introduce the logic for sequence_no processing. This is necessary to avoid the duplicate requests (possible when the master fails during the forward operations). The current protocol does not specify the logic. How ever the subcontract has the features/facilities to support sequence number usage/logic.

2. The logic for resolve, bind_new_context has to be changed. When the result returns a context back, we need to give a replicated context. The current code does not support it. How ever this should be an easy extension to do.

3. When the context has contexts that are implemented by some other domains, the current protocol will not work. We need to enhance the protocol. This is like, an object invoking the methods of some other object to satisfy (service) its current request (cascaded object invocation/ nested object invocation ???).

   There are two problems here.

   - The second (or n-th) level object can be replicated or it can be singleton. We need to handle these cases differently (?? we are not really sure of this).

   - When one calls from first level to second level (or from n-th level to n+1-th level, there will be duplicate requests, because all the replicas of first level will do this invoke (thus the replicas of second level will get the multiple invocations for the same "top-level" request.

   The paper [12] proposes a solution to this problem in message passing paradigm. We can convert this solution such that it uses only RPCs, but it may not be a very clean solution.

4. This point is mentioned in the paper. This is regarding the stale object handlers. We can avoid this problem by saving the object name in subcontract representation. When ever you find all stale pointers, get the new handler from the name server again. (this is for sobj example). As far as RNS goes, if all the pointers are stale, construct the repl_ctx again, as it is done in repl_ctx_mgr.cc