Naming Consistencies in Object Oriented Replicated Systems

Ganesha Beedubail and Udo Pooch

{ganeshb, pooch}@cs.tamu.edu

Department of Computer Science, Texas A&M University,
College Station, Tx - 77843-3112.

Technical Report (TR_96-007)
March 1996.

Abstract

In this paper we examine the naming consistency problems in distributed systems that support object replication. We define the meaning of naming consistency in these systems and observe that it is possible to have inconsistency in naming and yet have a consistent replicas of the object. It is also argued that the naming consistency is tightly coupled with the replica consistency protocols. We examine the properties of the naming consistency in some existing replicated object systems. Then we present and analyze the replica consistency protocols developed for Spring. These protocols allow more relaxed consistency requirements in the naming service and consequently the total cost of the object replication is reduced. Though these protocols are developed for object replication in Spring Operating system, the concepts can be applied to any general settings.

1 Introduction

A naming service (NS) is one of the central services of any distributed system. A well designed naming service provides the *location transparency* property to the objects in a distributed system. The NS provides *location-independent* names of objects for an application program by mapping the logical name to the object reference (which would encapsulate the physical address of the object). It is imperative that one must use a NS for developing distributed applications for insulating them from the effects of system reconfigurations. Another advantage of NS is that it provides an intuitive names for the objects instead of a string of bytes. The issues involved in providing the name service for distributed systems have been studied in the literature[1, 2, 3, 4]. In this paper we will discuss the issues involved in providing the NS for distributed systems that support object replication.

Object replication in distributed systems can be used for *fault tolerance (high availability)* and/or *load balancing* among other things. When an object 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.

The naming support required by the replicated object system is more complex than that required by the non-replicated object system. Specifically the name bindings at the name server and the instances of replicated objects should be mutually consistent. However, not much attention has been given to this area[5, 6]. In this paper we will define the consistency requirements of naming service (NS) for supporting replicated objects. We will observe that the naming consistency is tightly coupled to the *replica consistency* protocols. Then we will present the replica consistency protocols that provide more relaxed naming consistency features. These protocols are the enhancements of our previous work[7].

The rest of the paper is organized as follows. The next section briefly discusses the naming in distributed systems and explores the issues involved in *naming consistency* in distributed systems that support object replication. Section 3 we will discuss the related work on this issue. Specifically we will discusses *Reliable Distributed Programs* (RDP) architecture[6] and the *Arjuna* system[5, 8]. RDP and Arjuna systems describe the naming support for the object replication in the respective systems. Section 4 presents the modified replica consistency protocols for the object replication framework developed for Spring Operating System. It then analyzes the naming consistency properties of these protocols. Section 5 discusses and compares the naming consistency properties of the schemes presented in the prior sections. Section Section 6 discusses the future research and conclude this paper.

2 Naming Service in Distributed Systems

One needs *meaningful* names to identify the objects in a distributed system. The distributed name service (NS) provides this facility. The NS provides a mapping between the *names* and the *objects* (the name to object mapping is usually referred to as *name bindings*). More importantly, in distributed systems, the NS provides a means to locate the object in the distributed system. *Name bindings* (name to object mapping) in the NS usually contains the location information of the object. This information could be encoded in some implementation specific ways.

The semantics of the NS and the issues involved in designing and implementing NS for distributed systems are studied extensively. Since this aspect is not the primary focus of this paper, we will not discuss these issues here. The reader can refer to[1, 2, 3, 4] for the relevant reading in distributed name service.

2.1 Naming Consistency in Distributed Systems

First, we will discuss what we mean by *consistency*. We consider a system in which entities are related by some relation. We could say that the system is *consistent*, if the relation holds true amongst the entities at all times. In the case of a naming service, the name should correctly identify the object (the relation here is that, the name identifies the object). Thus a NS is consistent, if the name always points to the correct object (the object to which it was supposed to refer).

Consistency is not a problem in a static system (i.e., a system in which the system configuration does not change). In this case there is no possibility of inconsistency once the system was initialized in a consistent state. But a practical distributed system is never static. System components fail and recover, nodes are added and removed from the system, and objects move from one location to another. All these activities contribute to the possible inconsistent state of the system unless proper care has been taken. A well designed system tries to keep the system in a consistent state. Note that consistency can not be guaranteed at all times, since maintaining consistency in a large distributed system is an expensive task. Thus occasional inconsistencies do occur in the systems. It is for the application semantics to decide what kind of inconsistencies can be allowed and for how long the inconsistency can persist.

In this paper we are concerned with the consistency issues in the naming service for distributed systems that support object replication[1]. For the discussion in the rest of this paper we will make the following assumptions.

1. We assume an object oriented distributed system. The inter-object communication is carried out using remote procedure invocation. The processes in the system observe *fail-stop* behavior.
2. The system supports a distributed naming service (NS). The NS supported in the system is reliable (it could be made reliable by replicating it[9]).
3. 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).
4. When objects want to invoke the methods (operations) of other objects, they import the interface for that object from 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*.

2.1.1 Naming Inconsistencies in non-replicated systems

In distributed systems, when a client wants to invoke a service, it looks up in the name server the server's interface (object reference). Clients use this information for accessing the server. Ideally the client should lookup the interface for every remote invocation of a particular object. However, this adds to the name server lookup costs. A natural way of reducing the cost of name server lookups is

[1]This is different from the consistency issues in the *replicated naming service (RNS)*. Consistency in RNS is an issue where the NS is replicated for availability (fault tolerance) and load balancing. This problem is the same as the *replica consistency problem*[7, 8]. Consistency problems for RNS are studied in[9].

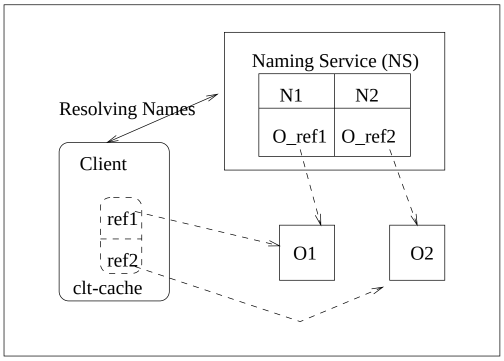

Figure 1: The client caches the name bindings for future use.

to have the clients cache the results of such lookups[6, 7]. Figure 1 shows this mechanism. Thus a client contacts the name server only when it imports the interface. It uses the same information for all subsequent remote invocations to that object. This raises the classic *cache invalidation* problem. That is, *what happens when a client's binding information becomes stale because the information at the name server has changed?*

The client's binding information can become stale for two reasons: (a) *the original server had died* (say object O1 has died), or b) *the original server had died and recovered* (say O1 died and recovered, in this case the O_ref1 at the NS will be different from the one at the client). For the client both these situations are identical. In the first case there is no server and in the second case the client cannot reach the server through its stale binding information. The present day *run time systems* are usually capable of detecting these situations. The client has to *re-import* the binding once it learns that the binding is stale.

Another way inconsistency can creep into the system is at the name server. What happens if a server dies after it exports its interface? The binding at the name server becomes *stale*. Following are some of the suggested solutions: 1) *Usage of a garbage collector:* A garbage collector here is a process that periodically checks all the registered bindings and the status of the respective servers. If it detects any failed server, then the corresponding binding can be deleted. 2) *Usage of invalidate procedure:* One can provide a special *invalidate* procedure for the name server. The client, upon detecting a crashed server, can invoke this method on the name service. Then the name server can delete the binding.

2.1.2 Naming Inconsistencies in Replicated Systems

First, we will discuss the meaning of *naming consistency* in the context of replicated system. It is hard to give a precise definition of *naming consistency* for a replicated system. The reason is that, the functionality (or the requirement) of the NS for replicated objects is not well defined. Many

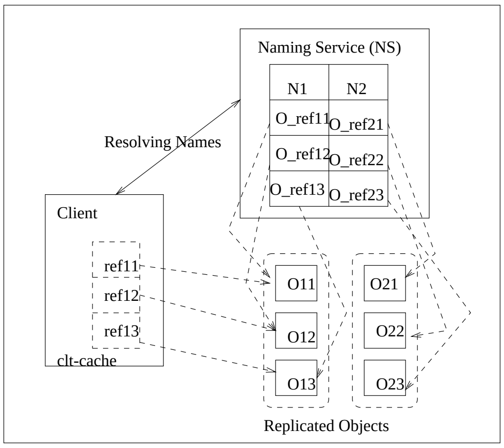

Figure 2: Naming Service requirements for systems that support object replication.

systems that discuss object replication do not talk about naming requirements for the replicated objects except for a few systems[6, 7, 8, 10].

The usual way of naming (representing) the replicated object is to associate a name to a list of object references. These object references point to the actual objects that constitute the replica set. The schemes presented in [6, 7, 8] are variations of this basic idea. Figure 2 shows the naming service requirement for a system that supports object replication. Now we will define the consistency requirement for the naming service:

- *The name-binding in a distributed system that supports replication is consistent provided that the object reference list associated with the name actually points to the correct (live) objects. When we introduce client caching, the object references at the clients must also point to the correct (live) objects. Also the list at the NS and at the client should agree with each other.*

This is a *strict consistency* requirement without considering the replication protocol and how the protocol makes use of the NS. Usually, the replication protocol used in replicated object systems are tightly coupled to a specific usage of NS. Thus the replication protocol may allow various relaxations for the above consistency requirement. Thus we can find that the name-binding is consistent for observable purposes, even when some of the object references are stale at the NS or at the client cache. Thus we will make the following observation.

- The effect of *naming inconsistency* in replicated object systems depends on the *replica consistency* protocol used. A name-binding that can be considered inconsistent in one replicated object system may not be considered so in another replicated object system.

3 Related Work

The technique of replicating the services (objects) for high availability has received much attention in the literature[6, 8, 11, 12, 13, 14, 15]. However except for a few[6, 8], these do not discuss the issues of object naming and binding support that is necessary for developing replicated objects. Few discuss the consistency issues and properties associated with the naming service for replicated objects. In the following we will examine the naming consistency with respect to object replication in two systems, namely: *Replicated Distributed Programs (RDP)*[6] and *Arjuna*[5, 8, 10]. Since it is essential to understand the replication algorithm to define the naming consistency in these systems, we will also briefly explain the replication algorithms used in these systems.

3.1 Replicated Distributed Programs (RDP)

Replicated Distributed Programs (RDP)[6] is a software architecture for fault tolerant distributed programs. This architecture has two basic mechanisms: (i) **Troupes:** A *Troupe* is a set of replicas of a module, executing on machines that have independent failure modes. Individual members of a *troupe* do not communicate among themselves, and are unaware of one another's existence. (ii) **Replicated Procedure Call :** is a generalization of *remote procedure call* for *many-to-many* communication between *troupes*. The semantics of *replicated procedure call* can be summarized as *exactly once* execution at all the replicas.

A client makes a replicated procedure call to the server *troupe* to invoke a service procedure. All the members of the server *troupe* execute this call in the same relative order (an optimistic *troupe commit* protocol ensures the same order at all the troupe members). Since the servers are deterministic, this ensures replica consistency. When a new (or a recovering) member joins the *troupe*, the state of the current server(s) is transferred to the joining member. The joining member also registers itself in the name service. Both of these activities happen inside an *atomic transaction.*

3.1.1 Naming Service in RDP

The client and server *troupe* use the *binding agent* (the name service in RDP) for importing and exporting the server interface names. The binding agent provides the following mechanisms in RDP:

- Facility to manipulate sets of module addresses and the facility to manage the troupe IDs required by the replicated procedure call algorithms.
- Facility to allow a third party to register the entire troupe.
- Facility to add or delete individual troupe in order to handle troupe reconfiguration.

3.1.2 Naming Consistencies in RDP

Now we will elaborate on the naming consistency problem and *client cache invalidation problem* in RDP. Let T be the set of members of a troupe, and let C be the cached set of members that a client believes constitutes the troupe. Then C is stale if and only if $C \neq T$. The possibilities for stale information correspond to the possible intersections of these two nonempty sets: (1) $T \cap C = \emptyset$, (2) $T \subset C$, (3) $T \supset C$, (4) $(T \cap C \neq \emptyset) \wedge (T \not\subset C) \wedge (T \not\supset C)$

The semantics of troupes and replicated calls require every member of a server troupe to execute a procedure if any member does. This will be the case if $T = C, T \cap C = \emptyset$, or $T \subset C$. The first two possibilities for stale information are therefore harmless. In the last two cases, however, the client calls some but not all of the troupe members. These calls will introduce replica consistency problem if they are allowed to succeed.

The solution proposed in RDP is to use troupe IDs as a form of incarnation number. Each call message carries the troupe ID of its destination as well as its source and each server troupe member rejects any call message whose destination troupe ID is incorrect. In RDP a troupe always changes both its membership and its troupe ID in an **atomic operation**. A server troupe member accepts a call from a client only if it bears the correct server troupe ID, which is the case only if the client knows the correct membership of that server troupe. In this case the naming is consistent.

3.2 Arjuna System

Arjuna[5, 8, 10] is a distributed system, implemented in C++, that provides facilities for constructing applications using persistent object which can be manipulated under the control of atomic actions (atomic transactions). Arjuna system provides several services for constructing fault tolerant distributed applications, namely: *atomic action service, RPC service, Object Storage service* and *Naming and binding service*. Here we are interested only in the object replication techniques and the naming consistencies for replicated objects.

3.2.1 Object Replication in Arjuna

Here we will briefly explain the object replication algorithm. Assume that an object A is replicated and a client want to access it (the client usually accesses the object A inside an atomic transaction). It involves the following sequence of activities: (1) creation of connections to persistent objects; (2) start of the atomic action; (3) method invocations: as a part of given invocation the object will be locked in read or write mode (assuming no lock conflict), and initialized if necessary with the latest committed state from the object store; (4) commit/abort of the action; and (5) breaking of the previously created connections.

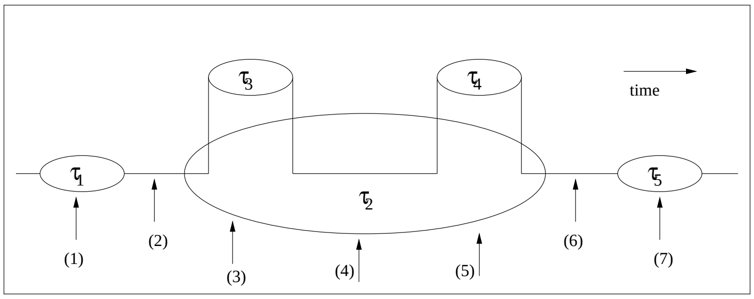

Figure 3: Accessing the GVD and the replicated objects in Arjuna.

A recovering node first completes any outstanding atomic actions whose phase two commit processing was interrupted by the crash. It then calls the *Recover* operations of the object server and state server databases to remove any entries kept for this node in any of the *use lists* (see below). This is necessary as these entries record pre-crash usage information that are now out-of-date.

3.2.2 The Group View Database (Naming Service)

The group view database (GVD) is used in Arjuna for maintaining the information about the replicated objects[5]. Here we will elaborate on the usage of GVD for object replication. The GVD is composed of two distinct persistent objects: an *Object Server* database (the *name server*) for maintaining *unique object identifier* to *object server* mappings for all the persistent objects and an *Object State* database for maintaining *unique object identifier* to *object state server* mappings. Clients use the object server database, while servers make use of the object state database.

3.2.3 Accessing the GVD (Naming Consistency)

In this subsection we will explain how the GVD operations are used by the clients while accessing a replicated object. The client operation is being executed as a top level atomic action. The operation contains calls to some object A (which is currently passive) that is replicated. Figure 3 shows the various stages in computation (indicated by the vertical arrows). It also explains how the consistency of GVD is maintained in the presence of failure of client and object server nodes.

Initially the client creates connections to the servers. This involves getting the object server information from GVD (*atomic action* τ_1) and using this information for creating the connections. The GVD is updated to reflect the fact that some servers are being used (by updating *use-lists* or had failed (if the client detects any failures). Then the servers are activated by obtaining the object state from the state servers. Again GVD is used (*nested atomic action* τ_3) for obtaining/updating the state server information. Then the application invocation takes place (*atomic action* τ_2). At the end of application processing the commit processing is carried out. During the commit process the GVD is updated (*nested action* τ_4) to reflect the object server/state server conditions (*failed/up-*

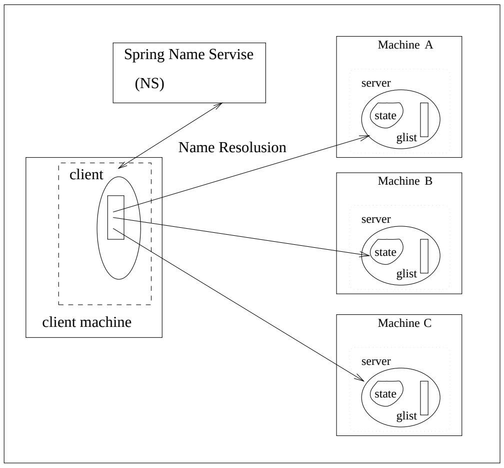

Figure 4: An overall view of the Replicated Object in Spring. The client obtains the glist form the Spring NS.

to-date). Finally the client breaks the connections by updating the GVD (*atomic action* τ_5). The inconsistencies created at the GVD by the client's failure during this whole activity is repaired only when the client recovers.

4 Naming Consistency in Spring for Object Replication

In this section we will present two replica consistency protocols used for object replication in Spring Operating System[16] and their naming consistency properties. These protocols are the modified versions of the protocols found in [7].

4.1 Object Replication in Spring

We developed an object replication framework[7] which was targeted to work on the Spring Operating System[16]. The framework is general; it can be used in any similar environment. Here we will discuss the salient features that are relevant in context with the naming consistency issues.

In Spring, we represent the replicated objects by a list of object references (we call this the *glist*). In our design, the objects are aware of the fact that they are replicated[2]. Hence each replicated object will also have the list of object references that contains the set of object replicas (i.e., every replica of a replicated object knows all other replicas of the same replicated object). In Spring, when we start a replicated object, we form the list of object references for all replicas (the *glist*). This *glist* is distributed to the replicas and then it is bound in the Spring Name Service[4]. Now any client that wants to access the replicated object can resolve the object name in the NS and can get the *glist*. It can then access the replicas using any replica consistency algorithm (the framework does

[2]Note that this is in contrast to the replication principle in RDP and Arjuna.

not restrict the replica consistency algorithm). Figure 4 shows an overall view of the replicated object in Spring.

4.2 Object Replication Protocol in Spring (Protocol-A)

In this section we will present one particular *replica-consistency* algorithm for the object replication in Spring. Recall that in Spring all the replicas and the NS have the *glist*. The client can get this *glist* from the NS (the client caches the *glist* from the NS). This protocol works in a *Master-Slave* mode. One of the replicas is designated as the master and the others will be slaves. A detailed protocol is given in [9]. We will just explain the high-level logic:

Client: The client sends a remote invocation to the master (it believes that the first server from its *glist*, that it can contact is the master). If it receives a result, then the invocation is complete. Otherwise it receives an object reference to the new master. Client updates it glist and repeats the process. If it cannot contact any of the servers (the whole client cache is invalid), then it gets glist from the NS.

Server: When a server receives a request, the following two cases arise: (a) It is the master: Then the server forwards the invocation to the slaves; performs the local operation; returns the result to the client. (b) It is a slave : If there is a current master, then this slave finds the object reference to the current master and returns it to the client. Otherwise it becomes the master and then this case is similar to the previous case.

A new (or a recovering) member that want to be the part of replica group follows the following protocol. The new member assumes the role of a client and invokes a join operation (*join_group()*) on the replicated object (for this it follows the client protocol explained above). At the master this results in the following:

- The master replica, which receives the *join_group* operation updates the glist and forwards the new glist to other members. It then transfers the state of the object (along with the glist) to the new member. Next, it updates the NS with new glist.

When the new member receives the current object and glist for its *join_group* invocation, it becomes part of the group.

4.2.1 Naming Consistency in Protocol-A

We will examine the above protocol to see whether there will be any inconsistency in the naming service. Obviously, if there are no failures and no addition of members, there will not be any inconsistencies. However even if there are no failures, if a new replica joins the group, then the client's *glist* will not have all the object references (note that the client cached the *glist* before the new replica joined the group). This situation need not be considered as an inconsistency however. This is because, the client accesses only one member in the replica set: the master. Even if it accesses any other object, it gets back the reference to the master. Thus as long as at least one of the object references in the client's glist points to a (live) correct server, the client can access the service of the object.

Now consider the situation at the NS and at the object replicas. Assume that one/some of the replicas crashed because of node failures. Then the glist at the NS and at the other replicas contains some stale pointers. Once again, this will not introduce any inconsistencies in the replicas (this will not cause any state divergence at the replicas). Thus this case need not be considered as a naming inconsistency (for the correctness of replica_consistency algorithm). However this situation introduces unnecessary overhead at the protocol level. Since the stale pointers are not removed from the glist, every time when a master forwards an invocation, it will try to contact a failed object (and thus a lot of time will be wasted in detecting that the replica had failed). Also, if a new client gets the glist from the NS it will have a partially correct glist and if the first replica in the glist had failed, then the client too will waste time while trying to invoke this replica (if that happens to be the master). Thus we would not want this kind of inconsistency in the naming.

The solution for this is simple. Whenever the master detects the failure of another replica (it will detect it while it forwards an invocation), it will update its glist. When the current forwarding is complete, it will forward this updated glist to other replicas. And finally it will update the NS. Note that updating the glist is an idempotent operation and multiple updates will not cause any harm.

Now consider the protocol for the addition/recovery of a replica. If no failure occurs during the addition/recovery process, then the naming will be consistent. If there is a failure of the replica that is being added to the system, then this reduces to the case discussed before and it can be solved similarly. Thus we will not have any naming consistency problem.

4.3 Object Replication Protocol in Spring (Protocol-B)

In this section we will present another *replica-consistency* algorithm for the object replication in Spring. The overview algorithm for replica consistency is as follows:

Client: The client sends the remote invocation request to all of the replica in its glist sequentially until it gets a reply for the invocation.

Server: When the server receives an invocation for an operation, it broadcasts the operation (invocation) to all other servers. This broadcast is an *ordered atomic* broadcast. When the broadcast is complete, it returns the results to the client.

The above protocol is an implementation of the *ordered atomic* broadcast from the clients to the server replica group. Since our servers are deterministic, this protocol ensures the replica consistency property[13].

A new (or a recovering) member that wants to be the part of the replica group follows the following protocol. The new member assumes the role of a client and invokes a join operation (*join_group()*) on the replicated object. This operation performs the following:

- The server replica, which receives the *join_group* operation reliably broadcasts (*ordered atomic broadcast*) the new glist to other members. It then transfers the state of the object (along with the glist) to the new member. Next, it updates the NS with new glist.

When the new member receives the current object state and glist for its *join_group* invocation, it becomes part of the group.

4.3.1 Naming Consistency in Protocol-B

The issues involved here is very similar to the discussion given in the context of protocol-A. All those situations and arguments apply here as well.

5 Comparison of naming consistency schemes

In this section we will review the naming consistencies provided by the three systems (RDP, Arjuna and Spring) and their relative merits. From the description of these systems it is clear that, the definition of naming consistency heavily depends on the replica consistency protocol. Also we can see that even though the naming was not globally consistent at all the times (i.e., the NS had some stale object references), it was not considered inconsistent for the particular system at hand. All we needed was that, the NS had to appear consistent to the application that was using the NS. Also we noticed that some inconsistencies only lead to performance problem rather than inconsistencies in the replicated object state.

We will now consider the cost of maintaining consistent NS for supporting replicated objects. The cost of naming consistency is associated with the replication (or replica consistency) algorithm. In effect this becomes a comparison of respective replication algorithms. However, some replica consistency algorithms allow relaxed consistency in NS (i.e., stale object references at the NS) without introducing the inconsistent replicated object state.

RDP : In this system, stale object references are allowed in NS as long as the client cache has the references to all the functioning server objects. This staleness leads to only performance problems. The client can update the NS once they detect the failure of a server[6]. We do not believe that permitting the NS update by the clients is a good practice. This is because it might lead to security problems (i.e., any arbitrary client can change the state of NS, it could do so maliciously). Another drawback of this system is that whenever a new (or a recovering) member wants to join to the *troupe*, the *troupe_id* has to be changed atomically (inside an atomic action). Also the incarnation number of the troupe changes and the client cache becomes invalid. The client has to contact the NS for a fresh *troupe* list.

The naming consistency required in RDP is conceptually simple, but it is relatively expensive to maintain when there is a lot of failure/recovery.

Arjuna : The situation in Arjuna is more complicated. Here the NS (GVD) is used in the replication algorithm in a complex manner. Along with the object reference, the GVD also maintains the *use list* (i.e., a kind of reference count: how many clients are using this object at this point?). A crash of a client leaves the *use list* entries in an inconsistent state. This is not a problem for normal operations. But if a new/recovering member wants to join the group, then this leads to problem. According to the Arjuna protocol, a new member can join only if the current members are in passive state (i.e., no one is using the current state of the object). But a crashed client leaves the object in *"being-used"* state until the client recovers. This requires that all crashed members/nodes eventually recover (practical considerations require that they recover at the earliest) to clean up the GVD. This is again not a good design, the system is susceptible to the weakest hardware in the system. Also GVD is updated by the clients, not good for security reasons.

In Arjuna, the replication protocol is complex and makes use of many (nested) atomic transactions. Consequently maintaining naming consistency is expensive. For example, a simple invocation of a 3-replicated object involves 9 atomic transactions (see the Figure 3: τ_3 and τ_4 are executed by all of the three replicas).

Spring : The major difference between the object replication in Spring and the other two systems is the way in which the naming of replicated objects is maintained. In Arjuna and RDP the names of replicated object is maintained at the NS and at the clients (cached at the clients). Whereas in Spring it is maintained at the NS, at the clients and at all of the replicas (the *glist*). This makes the replication algorithm simpler than the algorithm of RDP and Arjuna. Also it allows more relaxed consistency requirements at the NS. Note that Spring doesn't use any atomic transactions. Thus the replication protocol is less expensive.

In Spring, the client needs to have a correct object reference for only one member of the replicated object (this is true in both protocol-A and protocol-B). So the consistency required here is much simpler. Also in protocol-B the glist of the servers is changed by an *ordered atomic broadcast.* So they are always consistent. The NS need not have fully consistent *glist* for the similar argument we presented for the inconsistency at the client. However these inconsistencies may lead to performance degradation in protocol-B. But this will not happen in protocol-A, since the master object reference is given back to the client in response to a wrong invocation.

In Spring, the client does not update the NS or the glist maintained at the servers. Thus this protocol will not be as vulnerable as RDP or Arjuna for malicious clients. To summarize, we make the following observations:

- A consistent naming is necessary in a replicated object systems.
- The definition of consistency depends on the replica consistency protocol.
- Different protocols allow different degrees of inconsistency in the naming (still maintaining replica consistency). Since maintaining strict consistency is expensive, a protocol that allows more inconsistency in naming can be considered better. Also leaving inconsistency in naming should not cause much of a performance overhead (say, directing invocations to a failed server repeatedly).
- The replication protocol should avoid the necessity of a client updating the NS.

6 Conclusion

In this paper, we identified and defined naming consistency in distributed system that supports object replication. We examined the existing systems (RDP and Arjuna) that supports object replication and analyzed their naming consistency properties. The possible advantages of some feasible inconsistencies in NS are discussed. We also proposed some protocols for object replication in Spring, that allows high naming inconsistency (while maintaining the replicas of the application objects in a consistent state), thus reducing the total cost of replication. For future work it is necessary to formally define the naming consistency requirements for replicated object systems. Then, one can verify the various replication algorithms whether they satisfy the naming consistency requirements. The formalism of naming consistency problem may allow us to introduce maximum possible inconsistency at the NS. Thus the overall cost of replication can be lower. We also need to look closely at the interactions of replica consistency algorithms with the replicated naming service and identify the naming consistency problem. But we believe that our current observations should apply to this case as well.

References

[1] R. M. Needham, "Names," In *Distributed Systems*, S. Mullender, editor, pp. 89–101, Addison-Wesley, 1988.

[2] A. K. Yeo, K. L. Anada, and E. K. Koh, "A taxonomy of issues in name systems design and implementation," *Operating Systems Review*, vol. 27, no. 3, pp. 4–18, July 1993.

[3] J. J. Ordille, *Descriptive Name Services for Large Internets*, PhD thesis, Computer Science Dept., University of Wisconsin-Madison, 1993.

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

[5] M.C.Little, D.L.McCue, and S. Shrivastava, "Maintaining information about persistent replicated objects in a distributed system," In *International Conf. Distributed Computing Systems*, pp. 491–498, 1993.

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

[7] 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.

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

[9] 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.

[10] S. K. Shrivastava and D. L. McCue, "Structuring fault-tolerant object systems for modularity in a distributed environment," *IEEE Trans. Par. Distr. Syst.*, vol. 5, no. 4, pp. 421–432, April 1994.

[11] 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.

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

[13] 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.

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

[15] 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.

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