Fault Tolerant Object Replication Algorithm

Ganesha Beedubail, Anish Karmarkar and Udo Pooch

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

Technical Report (TR_95-042)
October 1995.

Abstract

This report describes a Fault Tolerant Object Replication Algorithm. The algorithm (protocol) uses Master/Slave configuration for replica consistency and fault tolerance. The novelty of this protocol is that, it is specified and implemented using only Remote Procedure Calls (RPCs). Lower layer (level) message passing mechanism is not used (which is usually difficult to implement and not easily portable on heterogeneous systems). To our knowledge this is the first protocol that is specified in terms of RPC only. This makes the implementation of the protocol very simple. Also the implementation is portable on heterogeneous systems which support a common RPC mechanism (like the DCE RPC). Another interesting feature of this protocol is that it is self contained. It does not assume the existence of any lower level reliable (like reliable communication of reliable group membership) services. However it assumes simple timeout mechanism for failure detection. The report also presents the proof of correctness of the algorithm and an analysis of the complexity of the algorithm.

1 Introduction

Object replication in distributed systems can be used 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.

As distributed systems deployed in practical use, RPC mechanism is becoming the preferred mode of communication since it is simpler to understand (an extrapolation of local procedure call) and simpler to program. Use of RPC implies that, all of the lower layer network communication details are handled by the stub mechanism. Thus many of the system vendors provide RPC support either directly in their Operating System (e.g.:SUN RPC) or as a middleware (e.g.: OSF DCE RPC). The design and implementation of RPC mechanisms under various contexts are discussed in [1, 2, 3, 4].

Since RPC is a well accepted communication mechanism in distributed systems[1] it is natural to expect to use RPC for implementing *fault tolerant distributed system protocols.* Of course, RPCs are implemented using the message passing facility of the underlying system. Thus with the help of appropriate message passing algorithms we can provide the fault tolerant RPCs[1, 4, 2]. But we feel that, it is not a good idea to provide fault tolerant RPC using message passing for the following reason. RPC mechanism is accepted because it is easier to develop distributed application using RPCs. If we have to implement FT RPCs using underlying system message passing mechanism we will loose one of the main advantages of RPCs. Also most of the current object oriented systems hide the underlying message passing mechanism from the user, thus making it difficult (almost impossible) for the programmer to directly handle the system messages. We faced this problem when we were developing a frame work for object replication in Spring[2][6]. Spring is one of the systems we encountered. The CORBA [7] framework is another such environment (if one is developing an application on CORBA, RPC is the only communication facility available). Thus it makes sense to explore the possibility of implementing fault tolerant replica consistency algorithm using only RPCs.

Our fault tolerant replica consistency algorithm is specified in terms of RPC only. Conceptually this is a simple extension of message passing algorithms. How ever, the practical implementation needs modification to the message passing algorithm. In message passing paradigm at the message destination, we usually separate the *message receipt* and *message delivery* events[3]. But in the case of RPC this is not possible because at the destination RPC mechanism invokes the server procedure (implemented by the user). The server state will be changed (permanently) when the procedure returns. Thus we need to keep enough information, while executing the procedure, so that this procedure can be executed in a different machine in case of a partial failure (when some members of the replicated object fails).

In our algorithm, processes communicate only when it is necessary (no periodic messages for detecting failures). Thus if there is no (useful) activity (client requests) in the system, no communication takes place, even when the servers fail. This is achieved by the active participation of the clients in the protocol. Thus this protocol give rise to minimum message transmissions in the system. Also maintaining the references to the active members of the server group (group membership protocol) is seamlessly integrated into the algorithm. It also integrates the re-joining of a replica to the replica group after a failure.

[1] Note that RPC is also the "Object Oriented' communication mechanism.

[2] Spring is an object oriented distributed operating system[5].

[3] This helps us to *re-order* the received messages to satisfy some kind of message order requirement.

1.1 Assumptions

We assume that a set of machines connected by a network. The machines in the network can fail only by crashing (*fail stop* assumption). The machine failure can be detected by *timeouts*. Further, the underlying RPC mechanism implements the *timeout* mechanism. If the RPC returns with a *timeout* then we assume that the target machine has failed. In the current work, network partition failure can not be tolerated but we are extending our protocol to do so. The underlying network may loose or duplicate the messages. But the RPC protocol will take care of these problems. We are also dealing with only non-persistent objects. We assume that, some machines of the server group (not necessarily the same machine, all the time) will always be up and will be having the latest version of the object.

We will also assume that the system does not have a distributed (global) name service. Thus the location of the replicated servers should be available to all the machines in the network. We will assume that a local machine name service is available in every machines that can be used to get the reference to the objects in that machine[4] In the following *server, object, and the process* are all used interchangeably.

The rest of the paper is organized as follows. The next section provides over view of the related work on object replication. Section 3 describes our system model. Section 4 describes our algorithm in detail. Section 5 provides the proof of correctness. Section 6 presents the message and time complexity for the algorithm. Section 7 discusses the future work and section 8 concludes the report.

2 Related Work

In this section we will briefly review the related research. Since this work is a special case of object replication, we will consider research 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*[8]. 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[5]. If the primary fails, then one of the backups becomes the primary. This approach is known as the *primary-backup* or the *primary copy*[9]. All the research work on

[4] Section 3 describes these assumptions in detail.

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

object replication that we are aware of are the variations or combinations of the above two basic strategies[6].

Alsberg and Day [9] 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 [10] Budhiraja et. al. present some theoretical aspects of primary backup approach. Their system assumes closely synchronized clocks in the system. They classify the primary-backup algorithms as *blocking* and *non-blocking* and provide lower bounds (under various failure conditions) on the *degree of replication, blocking time* and *failover time* for any primary-backup algorithm. The Harp File System also uses a primary-backup protocol[11]. 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 [12] 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 [13] 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[2] 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[7]. 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 sup-

[6]The voting scheme or variations of the voting schemes can be implemented as variations of primary-copy scheme.

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

ported in Amoeba to implement the directory service. The reader can refer to [17, 18, 19] for additional reading on replication.

3 System Model

Our algorithm is intended to work in a local area network (LAN). In principle, the algorithm will work in a wide area network (WAN). However the feasibility of such a scheme (i.e., object replication is a WAN) needs to be studied more carefully. The algorithm assumes that it does not have an access to a distributed naming service (in fact, we developed this algorithm to replicate naming service (NS) in Spring[6]). Thus, it is imperative that all the machines in the system know the location of the object servers (by some means, other than getting it from a name server). In our algorithm, we get the location of object servers from a file (may be configuration file) that is available to all the machines (we call this *ROSfile*). If it is impractical to have this file in all the machines, then only some machines in the network may have this file. Other machines can get this information by broadcasting a request. However, note that in the current network technology (bus based or ring based) the broadcast is limited to the IP subnets. Thus at least one machine in a subnet should have ROSfile. By not assuming a distributed name service our algorithm can be easily implemented in most of the current available systems that support an RPC service.

In our system the IP addresses or a machine uniquely identifies the machine in the network. We will differentiate between the *IP address* of a machine and the *object reference* of an object. Object reference is like a capability. If we have an object reference, we can access the object without knowing the location of the object (however the object reference would encapsulate the location of the object inside itself). We also need a local name service in the machines (a machine name service). This name service can be used to get the object references to the objects in that machine using a name (a *string* name). Thus we can get the *object reference* (capability) for an object in machine X, if we have the name for that object in machine X and the IP address of the machine X. Note that the *portmapper* of SUN RPC system is a local name service in this sense. Developing a local name service like this should be a straight forward exercise. We should also assume that the replicated objects will have a pre-defined *string* name. The following paragraph summarizes the foregoing discussion.

We assume a LAN that has M machines. We designate N of these M machines (N$\leq$M) as Replicated Object Server machines (ROS machines). Our replica consistency algorithm requires the identity of all the machines that host the replicated object servers. Thus we list all these N machines (the IP addresses of the machines) in a file (called ROSfile) in all the machines. 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.

```
typedef struct glist_type {
      int            IP_address;
      int            status; //UP or DOWN
      obj_ref_t      obj_ref; // Object reference
} glist_t;

glist_t glist[MAX_REPLICA];
```

Figure 1: Format of the Group List (glist).

4 Fault Tolerant Replica Consistency Algorithm

We use *Master-Slave*[13] configuration for replica consistency control and fault tolerance. The novelty of this algorithm is that: 1) It is self contained. It does not use any other services (other than the basic "*time-out*" mechanism to detect failures. 2) It is specified and implemented using only RPCs (synchronous communication). To our knowledge, this is the first protocol which does so. The following are the properties of the algorithm:

- All the machines in the system (the ROS servers and ROS clients) will have a group list (*glist*). The type(format) of each entry in this list is shown in figure 1.
- The order of the IP address list in the above list (glist) is fixed (and the same) in all the machines involved.
- *obj_ref* (pointer to the replica) and *status* changes as the machine go down and come up.
- 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.

In the following, we will describe the actual algorithm. The algorithm is presented in such a way that it can be implemented in any RPC system. In a general case we assume that the user implements the logic as specified in the algorithm[8]. For this paper we will show the fault tolerant replica consistency algorithm for any one *method*. The same algorithm can be implemented for all the methods.

[8]The algorithm can also be implemented by modifying the RPC protocol compiler (which generates the stubs) such that, the stubs include the algorithm implementation. Then it may become transparent to the client. But if the user do not want to modify the stub generator, then it can be implemented as shown in this paper.

```
invoke_remote_procedure( parms )
{
    seqNo = getSeqNo();// gets a systemwide unique sequence number
    i = 0; ret_code = NOT_OK;
    while (TRUE) {
        if (glist[i].status == UP) {
            ret_code = invoke_proc(glist[i].obj_ref, parms, seqNo);
            if (ret_code == OK) { return (results) };
            if (ret_code == TIMEOUT) { glist[i].status = DOWN; i++;}
            if (ret_code == NOT_A_MASTER) {
                master_id = get_id_from_results();
                glist[master_id].status = UP;
                glist[master_id].obj_ref = get_obj_ref_from_results();
                i = 0;
            }
        }
        else i++;
        if ( i ≥ MAX_REPLICAS ) { Rebuild Glist };
    } /* while (TRUE) */
}
```

Figure 2: Client Side Replica Consistency Protocol.

4.1 Client Side Algorithm

Figure 2 shows the part of replica consistency algorithm that is executed by the client (we assume that, the client invocation is implemented by *invoke_remote_procedure()* routine). 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.

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

2. It receives a *TIMEOUT* indication (a failure return code). The client simply invokes the next UP replica in its list (it also sets status of the current replica to DOWN).

3. It can receive a response which says that the target invoked is not the 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[9].

When all of the pointers are exhausted then the client should handle *stale pointer* problem. We can have *stale pointers* (in the glist) at the client side if all the original ROS servers go down and come up again (may be at different times). Thus the object references in glist becomes invalid.

[9] Note that all the replicas can not be out of service at one time, the assumption made in section 1.

The solution to this problem is to rebuild the glist. Once the client detects that all the references in its glist are invalid, it will rebuild the glist. It can do so with the help of IP addresses in the glist and a pre-determined object naming convention.

4.2 Server Side Algorithm

Figure 3 shows the part of replica consistency algorithm that is executed by the server. For implementing the server side of the algorithm, we need to do the following. For every *server_method()* we have to define/implement another method *server_method_forward()* with all the original parameters and three additional parameters as shown in the figure 3. We also need to implement a local object, called *ResultStore* (shown in the figure 3). This is used to store and retrieve the input/output parameters of a remote procedure call[10] identified by a (unique) sequence number (seqNo).

Explanation: All the servers maintain a counter called GUN (*Global Update Number*) which counts the number of updates performed to the server state since the initial server start up (time t=0). When the server procedure is invoked two cases arise: 1) It is not a master : This means that, the client either detected a failure of the master or the client's obj_ref to the master is stale. In either case, the server invokes *are_u_ok()*[11] method of servers, whose replica_identifiers are lower than this replica. If any of these servers return a YES, then that server should be the current master. Accordingly this server returns to the client the result (I_AM_NOT_A_MASTER and the obj_ref for the current master). If none of the invocations return a YES, then this server has to become the master. 2) This server is already a master.

The master server then forwards the invocation to the slaves (this is done by invoking *server_method_forward()* with appropriate parameters). The *dup_flag* is necessary for *server_method_forward()* to distinguish between (*the client retry because of a master crash*) and (*master crash, client crash and a new client request*). For the details of these scenario refer to figures 4, 5, 6 and 7. The server then performs the local operation (the actual server application logic), increments its GUN, saves the parameters and results in *ResultStore* object and returns from the invocation. Note that the update is performed by obtaining a *global lock*, thus the update operation at the server is serialized.

When a slave server gets invoked by *server_method_forward()*, with the help of *dup_flag* and *GUN*, it decides whether it missed any updates (a slave can miss an update, if the master fails during the forward operation and before retrying the request the client also fails, In this case some slaves may get updated and some may not). If it had missed any updates, it gets the parameters for that update

[10] This is necessary to tolerate the failure of the primary during the protocol execution.
[11] The object implementor should provide this method.

```
Int        I_am_master; /* YES if this replica is the current master */
Int        I_am_ok;      /* YES if Ready for servicing the client Request */
Int        replica_id;   /* position of this machine in RNSfile */
Int        GUN ;         /* global Update number */
Class      ResultStore {
           int       check_dup(seqNo);
           void      get_results(seqNo, results);
           void      store_results(seqNo, GUN, parms, results);
           void      get_parms(GUN,parms);
}
ResultStore          RS;

Server_Method(parms, seqNo)
{
     if (!I_am_master) {
          invoke are-u-ok() method on all servers whose ID is less than mine;
          if ( YES returned ) return (I-am-not-master, master_id, master_obj_ref);
     }
     /* If we come here, then I am the master */
     if (this_is_read_method) { execute_the_method(); return(results);}
     upd_lock();
     I_am_master = TRUE;
     dup_flag = RS.check_dup(seqNo);
     if (!dup_flag) Do_Update(SeqNo, GUN, parms, results);
     for (i=replica_id; i<MAX_REPLICAS; i++) {
          server_method_forward(glist[i].obj_ref, parms, seqNo, GUN, dup_flag)
     }
     if (dup_flag) RS.get_results(seqNo, results);
     upd_unlock();
     return (results);
} /* server_method() */

Server_Method_Forward(parms,seqNo, GunParm,dup_flag)
{
     if (dup_flag == YES) {
          if (RS.check_dup(seqNo)) return;
          else { Do_Update(SeqNo, GUN, parms, results); return;}
     }
     if (dup_flag == NO) {
          while(GUN < GUNParm) {//get all missed updates and update local state
               get_missed_update(parms, GUN);
               Do_Update(SeqNo, GUN, parms, results);
          } /*while() */
          Do_Update(SeqNo, GUN, parms, results);
          return();
     }
}

Do_Update(SeqNo,GUN,parms,results)
{
     invoke_local_method(parms); /* the actual logic of the method */
     GUN ++;
     RS.store_results(SeqNo, GUN, parms, results);
}
```

Figure 3: Server Side Replica Consistency Protocol.

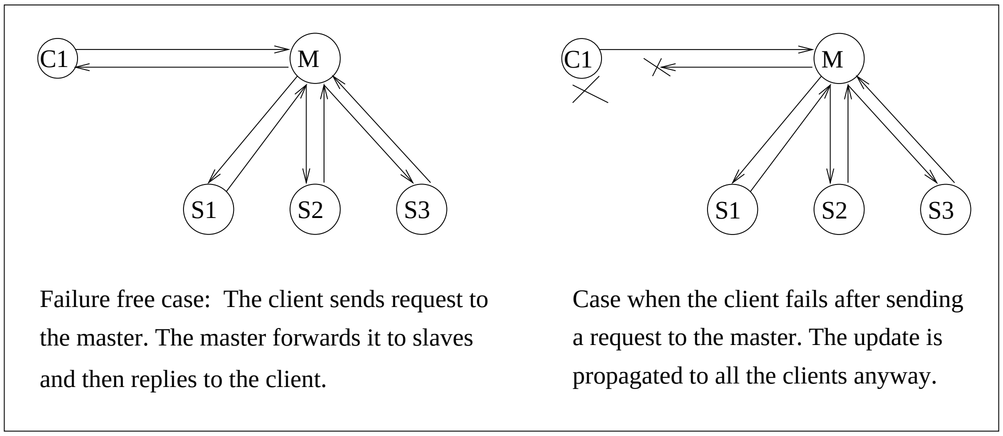

Figure 4:

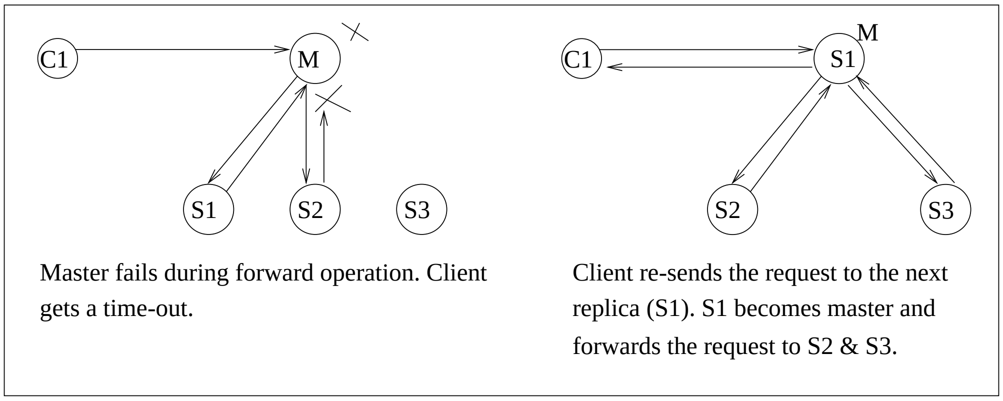

Figure 5:

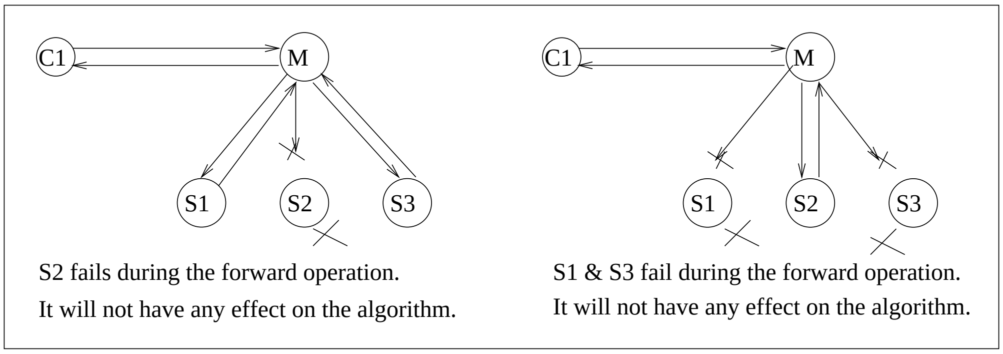

Figure 6:

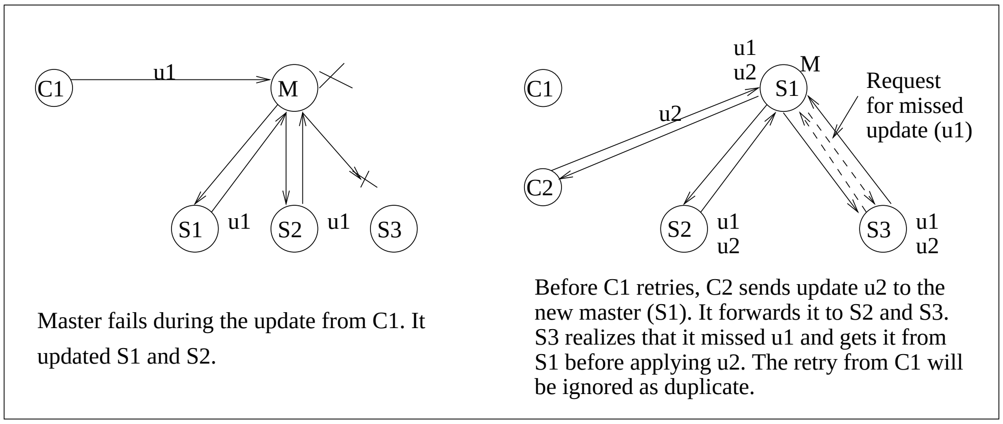

Figure 7:

from the current master (which should have it in its ResultStore Object) using *get_missed_update()*. It then performs appropriate updates.

4.3 Replica Join Protocol (Server Start up Algorithm)

Figure 8 shows the algorithm executed by the ROS server when it first starts up (i.e., when the machine comes up). The same protocol is executed when it starts up after a failure.

Explanation: The server first gets its IP address from the system and fills its group list (from the RNSfile, see section 3.3). It then determines, its replica_id (its position in the group list). It then creates a local object and binds it in the local machine name space. It then gets the object references (obj_ref) for other servers (it can do so using the IP address and pre-defined naming conventions). Then, it finds out the first running server in its group list. If that server is itself, then it becomes the master, otherwise it becomes the slave. To become the master, this server will contact the current master. The current master transfers the object state[12] to this server and gives up its *mastership* (by resetting its I_am_master flag). To become a slave, this server simply gets the object state from the current master.

5 Proof of Correctness

In this section we will prove that the algorithm given in the previous section is correct. First we need to define the *correctness*. Here we will assume that replication is done for fault tolerance. If we could assume that there would be no failures in the system then we would not need the replication. Thus the user would interact with only one object. But in practice we will have failures in the system. To mask the failures we replicate the object. Since the purpose of replication is to provide

[12] The object implementor should provide a method to do the object state transfer.

```
main()
{
      get_my_ip_addr();
      total_servers = fill_glist(); // from ROSfile
      replica_id = get_my_replica_id(); // my position in ROSfile

      upd_lock();
      I_am_ok = NO;
      create_and_bind_new_object();
      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;
      } /*while */
      upd_unlock();
}

Become_Master (int idx)
{
      if (I_am_the_only_one_UP) { I_am_master = YES; return(OK);}
      for ( i = my_replica_id+1; i < MAX_REPLICAS; i++) {
            ret = join_the_group(glist[i].obj_ref, my_idx, my_obj_ref);
            if (ret == NOT_READY) continue;
            if (ret == I_AM_NOT_MASTER) return(NOT_OK);
            if (ret == NOT_OK) return(NOT_OK);
            if (ret == OK) break;
      }
      return(OK);
}

Become_Slave (int idx)
{
      for ( i = 0; i < my_replica_id; i++) {
            ret = join_the_group(glist[i].obj_ref, my_idx, my_obj_ref);
            if (ret == NOT_READY) { PAUSE; return (NOT_OK);}
            if (ret == I_AM_NOT_MASTER) return(NOT_OK);
            if (ret == OK) return(OK);
      }
      return(NOT_OK);
}
```

Figure 8: Server Start up Protocol

```
Join_the_group(dst_idx,dst_obj_ref)
{
    /*Note: Join is an Idempotent Operation, thus no seqNo logic is necessary.
    if (! I_am_ok) return (NOT_READY);
    upd_lock();
    for ( i = my_replica_id+1; i < MAX_REPLICAS; i++) {
        ret = join_forward(glist[i].obj_ref, dst_idx, dst_obj_ref);
    }
    glist[dst_idx].obj_ref = dst_obj_ref;
    glist[dst_idx].status = UP;
    State_Transfer (dst_obj_ref);
    if (dst_idx < my_replica_id) I_am_master = NO;
    upd_unlock();
    return(OK);
}

Join_forward (int dst_idx, dst_obj_ref)
{
    glist[dst_idx].obj_ref = dst_obj_ref;
    glist[dst_idx].status = UP;
}
```

Figure 9: Replica Join and State Transfer Protocol

tolerance against failures, the replication should not be visible at the user or action level[20]. For performing actions, it should appear as if there is a single copy of the object. An action will perform operations on logical copy of the object and the underlying system (algorithm) should map it to operations on multiple copies of the objects. To be correct, the mapping must ensure that the concurrent execution of actions on replicated object is equivalent to a serial execution of actions on non-replicated object. That is, the execution should be equivalent to a serial execution on non-replicated object. This correctness property is called *one copy serializability* (1SR). Following is a definition of 1SR[15].

Definition 1 *One Copy Serializability (1SR): If the effect of a group of [atomic] actions executing on a replicated object is equivalent to running those same [atomic] actions on a single copy of the object then the overall execution is said to be one copy serializable (1SR).*

The algorithms to manage replicated objects such that 1SR is satisfied are called *replica control algorithms.* 1SR requires that the different copies of objects must be in mutually consistent state so that the user actions get the same view of the object state. Due to this consistency requirement, replica control algorithms are also called *consistency control algorithms.* As we can see, if the algorithm can ensure 1SR then the algorithm also ensures the mutual consistency of replicas. Following definition restates this[15].

Definition 2 *A replica consistency protocol is one, which ensures 1SR.*

Outline of Proof: We use the *master-slave* configuration. Here the clients always contact the master. Hence clients always see one copy of the object. Thus to prove that our replica control algorithm is 1SR, we should show:

1. There will be only one master in the system.

2. Updates are not lost due to (partial) failures.

Note that, there need to be only one master in the system when any client wants to access the object. If the system is in idle state (i.e., no client accesses the object) and if the current master dies, we need not elect (create) a new master immediately. Only when a client accesses the object next time there should be a master in the system. Thus the statement 1 above can be modified as *there will be utmost one master in the system at any time.* In the following, *lemma 2* proves this aspect. *Lemma 1* is used for proving the *lemma 1.* *Lemma 3* states the *update semantics* of the algorithm which satisfies the statement 2 above.

Lemma 1 *At any time t when two servers S_i and S_j are UP, both of them can not decide that it the only server that is UP.*

Proof: Assume that at a time t when both S_i and S_j are UP, both S_i and S_j decide that it the only server that is UP in the system. Then the *glist status* vector for S_i will be : status[i] = 1 and status[k] = 0 (for $k \neq i$) and the status vector for S_j will be: status[j] = 1 and status[k] = 0 (for $k \neq j$)

Now we can infer from figure 9 (note : $\rightarrow$: happened before), for S_i:

$$S_i CreatesObj \rightarrow S_i InvokesGetObjRef \rightarrow S_j RepliesNegative \rightarrow S_i SetsStatus[j] = 0$$

For S_j:

$$S_j CreatesObj \rightarrow S_j InvokesGetObjRef \rightarrow S_i RepliesNegative \rightarrow S_j SetsStatus[i] = 0$$

Also

$$S_j RepliesNegative \rightarrow S_j CreatesObj$$
$$S_i RepliesNegative \rightarrow S_i CreatesObj$$

From the above we can have :

$$S_i RepliesNegative \rightarrow S_i CreateObj \rightarrow S_i InvokesGetObjRef \rightarrow$$
$$S_j RepliesNegative \rightarrow S_j Createsobj \rightarrow S_j InvokesGetObjRef \rightarrow$$
$$S_i RepliesNegative \rightarrow S_i CreatesObj$$

We have a Cycle here, thus the contradiction. Hence the lemma is proved.

Lemma 2 *At any time t there will be ≤ 1 master in the system.*

Proof:

We will prove that there can not be two masters in the system at any time t. Similar argument can be extended to $n > 2$ master case.

Assume that at time $t \geq 0$ there are two masters (S_i and S_j) in the system and ($i < j$). The events that leads any server S_k to become master are the following.

A) WhenS_k startsup :

R1: No other server is UP

R2: S_k's replica_id (k) is the lowest among the servers those are UP

B) When S_k is running as a Slave :

R3: S_k receives and Update request and all the Probe(*are_you_ok()*) invocations return NO.

Now we need to consider the following cases:

Case 1: S_i became master because of R1. This implies :
S_j cannot become master because of R1 (S_i is UP, we proved this case in *lemma 1*).
S_j cannot become master because of R2 ($i < j$).
S_j cannot become master because of R3 (S_i will return an OK for the probe message).

Case 2: S_i became master because of R2. This implies that S_j cannot become master because of the reasons given for case 1.

Case 3: S_i became master because of R3. This implies that S_j cannot become master because of the reasons given for case 1.

Case 4: S_j became master because of R1. This implies :
S_i cannot become master because of R1 (S_j is UP).
S_i can become master because of R2. Then S_i invokes Join_group() (When S_j is master). This implies that the join method in S_j will reset the *I_am_master* flag for S_j. Thus there will be only one master left.
S_i cannot become master because of R3 (because S_i will not run as a Slave, since $i < j$).

Case 5: S_j became master because of R2. This implies that there will be only one master for the reasons given in case 4.

Case 6: S_j became master because of R3. This implies that there will be only one master for the reasons given in case 4.

Thus we will not have two masters in the system at any time. Hence the lemma is proved.

Lemma 3 *Following is the Update Semantics guaranteed by the algorithm:*

A: *If the client does not fail during the update, then all the UP servers will receive the update (in the same order).*

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

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

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

Proof:
(A): We consider two cases, when the client does not fail during the update :

Case 1: The current master does not fail. Then the master forwards the update to all the slaves (servers).

Case 2: The current master fails during the update/forwarding operation. Then the client will try the next UP server in its *glist* (which will become master or the client will get the reference to the new master). and thus the update/forwarding operation gets completed.

Thus, if the client does not fail during the update operation all the servers will receive the updates.

(B): We consider two cases, when the client fails during the update :

Case 1: The current master does not fail and it gets the clients update invocation. Then the master will complete the update/forward operation and thus all the UP servers get receive the update.

Case 2: The master fails during the update/forward operation. Let the current request from the client (say C_1) be RQ1. Consider the following cases:

Case 2.1: None of the slaves got the updates, i.e., the master failed before any forward operation. In this case the update is not recorded in any of the servers.

Case 2.2: Some of the slaves got the update, i.e., the master failed after some forward operation. But those servers too failed before the receipt of next update request from any client. In this case too, the update is not recorded in any of the surviving servers.

Case 2.3: Some slaves got the update (RQ1). Now some other client (say C_2) sends an update request (RQ2). This RQ2 will be processed by the new master (say S_i is the new master, S_i is the running server, whose replica_id, i is the lowest among the running servers). Then either S_i had received RQ1 or not. If it did not receive RQ1 then this case is similar to case 2.2 . Is it had received RQ1 then, while processing the forwarded operation, servers (slaves) S_j ($J > i$) will recognize that they did not get RQ1 (with the help of GUN). Thus they will get RQ1 from S_i and will process it first before processing RQ2. Thus eventually all the UP servers will get RQ1.

(C): From the algorithm, we can see that (both at the master and at the slave) an update operation increments the GUN (which starts at 0). This implies that every update (to the object) can be uniquely identifiable (by GUN). In the algorithm, the master determines this identifier (for any update), and when it is forwarded to the client, it gets the same update identifier. Had the slave missed some updates due to failures (which is determined by comparing slave's current local GUN and the GUN associated with the current forwarded update) it obtains those updates first (from the current master). Thus all the *up servers* apply the updates in the same order.

Lemma 4 *Following is the read semantics guaranteed by the algorithm.*

Any client that reads the object gets the latest state of the object (any client that reads the object gets the value written immediately before this read).

Proof: Note that the client reads from the current master and all updates are carried out at the master first and then forwarded to the slaves. Thus a client will always read the latest state. Note that if the master fails during the forwarding phase, some of the slaves may not have the latest update.

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

Proof: This follows from lemma 2, 3 and 4. Lemma 2 and 4 ensures that the algorithm is 1SR (because the effect of atomic actions acting on replicas is the same as the effect of those atomic actions acting on the master). Lemma 3 and 4 ensure that if the client does not fail during the operation, the operation succeeds (i.e., the client sees a fault tolerant server).

6 Complexity Analysis

In this section we will analyze the complexity of our algorithm. We will consider two complexity measures:

Message Complexity: This is the number of messages used (transmitted) for an update or read request. We consider one RPC is equivalent to two messages (one request + one response).

Time Complexity: This basically measures the elapsed time between the start of a request and the end of the corresponding response seen by the client. We measure this in terms of time for single message transmission over the network (let it be δ_t) and the time for processing an update request (let it be δ_w) and the time for processing a read request (let it be δ_r). The δ_t is the time required to transmit a message from one node (say) client to another node (say server) on a point-to-point network. For simplicity sake we assume that $\delta_r = \delta_w$ and $\delta_t + \delta_w + \delta_t = \delta$.

We assume that we have n replicas (one master and $(n-1)$ slaves).

1. Failure free message complexity : $2n$ messages (2 messages between the client and the master, $2(n-1)$ messages between master and the slaves).

2. Failure free time complexity (for write):

   $$\delta_t + \delta_w + (\delta_t + \delta_w + delta_t)(n-1) + \delta_t = (n)(2\delta_t + \delta_w) = (n)(\delta)$$

3. Failure free time complexity (for read):

   $$2\delta_t + \delta_r = \delta$$

   The analysis of message complexity in the presence of concurrent failures of the servers is more difficult. However, for simplicity we will assume that single failures are more likely. Let the *time-out* period be Δ and $(\Delta >> \delta_t or \delta_w or \delta_r)$.

4. Message complexity when the master fails (for write):

   $$1 + 1 + 1 + 2(n-2) = 2n - 1$$

   Here we assume that, we send only one request message and a *time-out* instead of reply message implies the failure of the current master. Then we send the request to the next server in the glist.

   Similarly, message complexity when one slave fails $= 2n - 1$.

5. Time complexity when the master fails (for write):

$$\delta_t + \Delta + \delta_t + \Delta + \delta_w + \delta_t + (n-2)(\delta_t + \delta_w + \delta_t) = 2\Delta + (n-1)(\delta) + \delta_t = 2\Delta \text{ (since } \Delta >> \delta)$$

The time complexity when a slave fails is Δ. Thus the time complexity lies between Δ and $(n-1)\Delta$ in case of failure.

7 Discussion of Future Work

In the current form, our algorithm requires that every host machine know the identity of all the machines that host the replica of the object. This fixes the server machines apriori. The advantage of this scheme is that the algorithm does not need a name service. In fact we used this algorithm to develop a *replicated name service* for the Spring Operating System. This is also satisfactory for applications where the host machines for object can be decided before hand. However, this restriction may not be acceptable in a general environment. We need to modify the algorithm such that, the host machine for the objects can be added/deleted/replaced for the replicated objects dynamically. We may have to use a *fault tolerant naming service* (using the current form of the algorithm) to achieve this effect.

The current algorithm does not tolerate network partitions. However, a simple extension (inclusion of *majority voting scheme*) of the algorithm will solve this problem. But this will increase the time and message complexity for read operation.

Note that the *time complexity* of our algorithm is *linear* with the number of replicas. Since, update operations can be done in parallel (simultaneously) in all the replicas, this time can be reduced. This can be done either by using a *hardware multicast* or using multiple threads to forward the requests (concurrently) at the master. However, this needs modification to the current algorithm such that when a slave becomes master, it should contact all other slaves, so that it will get this latest update.
In the current implementation, we did not address the issue of *garbage collection* in the *ResultStore* object. Note that, every invocation to the replica stores its parameters/results in the ResultStore. We need to delete this information when it is no longer useful. One simple solution is to periodically delete all *old* (older than a pre-determined time value) information (a *thread* running in ObjectStore can do this). Also, note that we need to keep the information related to the latest invocation from a client. All prior information can be deleted.

We said that the replication can be used for load balancing. But in our algorithm, we always send the requests to the master. This is to achieve *strict sequential consistency*. If we can relax this condition, then the read requests can be sent to any replica. Or we need to modify the algorithm such that more concurrency is possible for read operation. We are investigating this possibility.

8 Conclusion

We have presented a simple algorithm for implementing fault tolerant objects in a distributed system. The algorithm uses master-slave configuration. It is specified only in terms of RPCs. This will simplify the implementation of complex objects substantially (by using the available support for RPCs in a general setup). A preliminary implementation of this algorithm is carried out to provide aa replicated name service for the Spring Operating System. The implementation details and the performance figures can be found in[6].

References

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

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

[3] A. D. Birrel and B. J. Nelson, "Implementing remote procedure calls," *ACM Transactions on Computer Systesm*, vol. 2, no. 1, pp. 39–59, February 1984.

[4] M. D. Wood, "Replicated rpc using amoeba closed group communication," In *International Conf. Distributed Computing Systems*, pp. 499–507, 1993.

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

[6] 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.

[7] T. J. Mowbray and R. Zahavi, *The Essential Corba: System Integration Using Distributed Objects*, John Wiley & Sons, Inc., 1994.

[8] 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.

[9] 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.

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

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

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

[13] G. Beedubail et al., "Fault tolerant objects in distributed systems using hot replication," Technical Report TR95-023, Computer Science Department,Texas A&M University, April 1995.

[14] T. Joseph and K. Birman, "Exploiting 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*, pp. 130–139, 1993.

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

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

[19] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle, "Fault tolerance under UNIX," *ACM Trans. Comp. Syst.*, vol. 7, no. 1, pp. 1–24, February 1989.

[20] P. Jalote, *Fault Tolerance in Distributed Systems*, Prentice Hall, 1994.