Fault Tolerant Objects in Distributed Systems Using Hot Replication

Ganesha Beedubail, Anish Karmarkar, Anil Gurijala, Willis Marti and Udo Pooch

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

Technical Report (TR_95-023)
April 1995[1].

Abstract

This paper presents a new algorithm for supporting fault tolerant objects in distributed systems. The fault tolerance provided by the algorithm is fully user transparent. The algorithm uses a variation of object replication scheme, which we call the Hot Replication Scheme. The algorithm supports nested object invocations. The chief advantages of the scheme are : a) No action is needed in the case of failure of a secondary replica, b) The time to recover from a primary failure is minimal, c) Separation of replication protocol and reliable communication protocol. To recover from a primary failure the system need to (detect the failure and) select one of the secondaries to become the primary. The designated secondary can become primary once it has made sure that its current state is equivalent to the state of the failed primary (it can do so by processing outstanding requests, if any). This is in contrast with the checkpointing and rollback recovery scheme, where the recovery time can be substantial. Our Algorithm exploits the general features and concepts associated with the notion of the objects and object interactions to its advantage.

1 Introduction

Fault Tolerance provides value addition for distributed systems. When compared to the centralized systems, distributed systems have inherently more fault tolerant capability. This is because in distributed systems, the computing elements and other resources are physically distributed and thus there is no single point of failure. One can always expect one or the other part of the system to be up and running and available for service. However, realizing this expectation in practice is not trivial. Providing uninterrupted service (fault tolerant service) in a distributed system is a challenging task for the designer of such a system. In this paper, we will describe an algorithm (methodology) for supporting fault tolerant services in distributed systems.

[1]This Paper was revised in December 1995. The revised version appears in the Proceedings of the Fifteenth International Phoenix Conference on Computers and Communications (IPCCC'96), Phoenix, AZ, March 1996.

Various schemes for providing fault tolerant services in distributed systems do exist in the literature[1, 2, 3, 4, 5, 6]. However, most of these systems use process and message passing models of distributed systems. But currently the object oriented technology is getting widespread acceptance (for many of the advantages seen in this technology), in the design of distributed systems. Thus in this paper we will consider the object oriented model (in which the objects interact by remote method invocation) of the the distributed system. In this case we can make use of the general characteristics of the *objects* and *synchronous remote method invocation* for designing efficient algorithms for providing fault tolerant distributed objects (services). We will provide the fault tolerant objects by replicating the objects (we call this method as *hot replication scheme*) on hosts that fail independently.

This work is done in the context of the Phoenix project, that is currently being carried out at our Distributed System Laboratory. The Phoenix system will provide fault tolerance capability for distributed object oriented systems. The major (inter dependent) components of the Phoenix system are: fault tolerance support for distributed objects (provided by FTO Layer), fault tolerant naming service (FTNS), fault tolerant communication services (provided by FTC Layer) and distributed system health checking (distributed system diagnosis). Phoenix provides fault tolerance at two levels: *user (application) transparent* and *user (application) assisted*. In the first case, the system provides fault tolerance for any application that is developed (as an object oriented distributed application) without considering that faults could occur in the system. In the second case, users specifically develop the application to tolerate the faults using the library functions provided by Phoenix (application programmers interface - APIs). As can be expected, the transparent fault tolerance support will be more expensive compared to the explicit fault tolerance provided by the services (objects) developed using APIs.

The next section briefly discusses the related work in this area. Section 3 describes the model of the system and assumptions about the system. Section 4 presents our algorithm for supporting fault tolerant objects in distributed systems. Section 5 concludes the paper and discusses future work.

2 Related Work

Fault Tolerant services are provided by two different approaches. In the *Checkpointing and Rollback Recovery* approach, the state of the system is checkpointed in stable storage as necessary. If failure occurs during the lifetime of the service, the prior system state is restored from the stable storage and the execution is continued from that point onwards[2]. In the *Object Replication* or *Modular redundancy* approach, multiple copies (instances) of the object (service)

[2]We will not go into the details of this approach. For additional details on checkpointing the reader can refer to [7].

are used to provide the resiliency against host failures.

Object Replication (or Modular Redundancy) approach for fault tolerance generally assumes a *client-server* based distributed system. The server is (usually) replicated. In this approach, the copies of the objects are usually arranged in two main schemes: *primary backup* and *active replication*. In the case of *primary backup* scheme, one of the replica is considered as the primary and the other replicas as backups or secondaries[8, 9, 10, 11]. Only the primary interacts with the client and other objects. The primary propagates all the necessary information (request/reply messages) to the secondary, so that the secondaries are kept synchronized with the primary. In the case of *active replication* scheme, all the copies of the objects execute the client request simultaneously and return the result to the client[2, 3, 12, 13, 14]). The client selects the results depending on the assumed failure modes.

Alsberg and Day [9] proposed one of the earliest primary-backup protocols. The paper presents the error detection and recovery schemes for *two-host resiliency* (tolerating one failure) and the extension of this concept to n hosts. In [10], Budhiraja et. al. present some theoretical aspects of the primary backup approach. Their system assumes closely synchronized clocks in the system. They classify the primary-backup algorithms as *blocking* and *non-blocking* and provide lower bounds (under various failure conditions) on the *degree of replication, blocking time* and *failover time* for any primary-backup algorithm.

In [8], Birman et. al. describe a technique to implement *k-resilient* distributed objects. They use a *coordinator-cohort* scheme, a variation of the 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 the scheme proposed by Yap et.al[11] for implementing the fault tolerant remote procedure call, the replicas (also called incarnations) are organized as a chain. The first replica acts as the primary, the second replica acts as the backup for to the primary. The replica (incarnation) $i + 1$ acts as a backup for replica (incarnation) i. The primary forwards the request and reply/results it gets to its secondary/backup.

In [13], Fred Schneider presents a detailed model, which he calls a *state machine approach* for implementing fault tolerant services. They discuss the message agreement and message order requirements for this method. Eric Cooper describes[2] replicated procedure call (RPC) mechanisms for constructing highly available distributed programs. Replicated remote procedure call combines RPC with replication of a program module for providing fault tolerance. In [12], Jalote describes an algorithm for resilient objects in broadcast networks, which reduces the number of messages required to keep copies of the object in a consistent state. The Arjuna system uses the facilities of atomic transactions and naming and binding services for providing the replicated objects[14].

The active replication approach for fault tolerance gets complicated when objects can make nested invocations of other objects. Consider a client making a request (top level request) to a replicated object (say object A). This request give rise to another request from object A to another object B (maybe replicated). Now all the replicas of object A generate this request to object B. These requests are called the images[12] of independent requests (toplevel requests). At object B only one of these images should get executed. The scheme[12] proposed by Jalote, uses a unique identifier for the same images (image set) of the independent top level requests. Only one image having this unique identifier is executed.

In the scheme we propose, one replica is considered as the primary through which all communications outside the replica group takes place. Other copies (replicas) are secondaries. The primary reliably multicasts all the requests and replies it gets from the clients and other objects to its secondaries (backups). Because of this the secondary replicas know the current state of the primary. The secondaries remain active and execute the requests forwarded from the primaries just like the primary execute the client requests. Thus we call the secondaries as *hot replicas*. Since we arrange the replicas in primary backup scheme, the problem of image requests as explained above will not arise here.

3 System Model

Our system, in which we support fault tolerant objects, is a distributed object oriented system. This system is a collection of objects that provide services to the user. The objects interact with each other and with the outside world through well defined interfaces. The object interfaces are defined in such a system using some standard interface definition language (IDL), though this fact is not necessary (or assumed). Systems such as SPRING[15, 16] provide such features. The objects in the system are active (i.e. the object is always executing for providing services).

Objects basically contain the data (also called object state) and code (procedure or methods) to manipulate the object state. The object data can only be accessed by invoking the object methods (or object interfaces). Any object that is accessible by the user (or other objects) has publicly known interfaces. Objects announce their service availability to the world through some kind of naming (or directory) service. Once activated (the execution of object starts), the objects announce their service (by registering in a name service) and wait for requests from the users (clients). When a request arrives, the object services the request, sends the response and then waits for another request. For serving a request the object may invoke the services

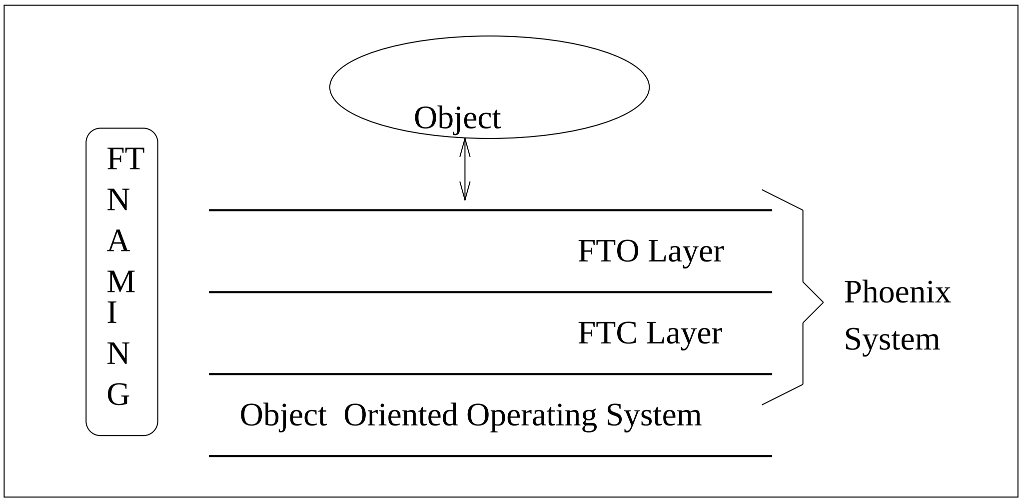

Figure 1: The System Model.

of other objects. So when an object sends an invocation to another object, the invoking object blocks until it receives a reply (synchronous communication). This object invocation can nest to any level (nested object invocation).

We assume that it is possible to distinguish the client request message to the object (and response message to client) from any other messages arriving (and leaving) from the object. This distinction is made without knowing the semantics and logic of the object but just by examining the contents of the message header. This is a reasonable assumption, since objects interact through a well defined protocol that is known to the system but transparent to the user (application). Thus all the objects (without regard to the object logic) use the same protocol for inter object communication. We categorize the messages into the following four types: *CL-REQUEST* (Object service request message from the client to the object), *CL-RESPONSE* (Object service response message from the object to client), *OBJ-REQUEST* (Object sends this request to the other objects) and *OBJ-RESPONSE* (Object receives this message from the other objects). Note that *OBJ-REQUEST (OBJ-RESPONSE)* is a client request (response) for another object.

For the purpose of our algorithm we assume that the system provides reliable point to point communication and reliable atomic multicast services. A distributed failure detection mechanism is present in the system which detects the failures of the nodes (and thus the objects) and invokes the object recovery protocol. We also assume the existence of a fault tolerant naming service. In Phoenix system these services are provided by the various Phoenix components. Figure 1 shows the assumed model of the system. We assume that the nodes in the system can fail by crashing (i.e. fail stop assumption).

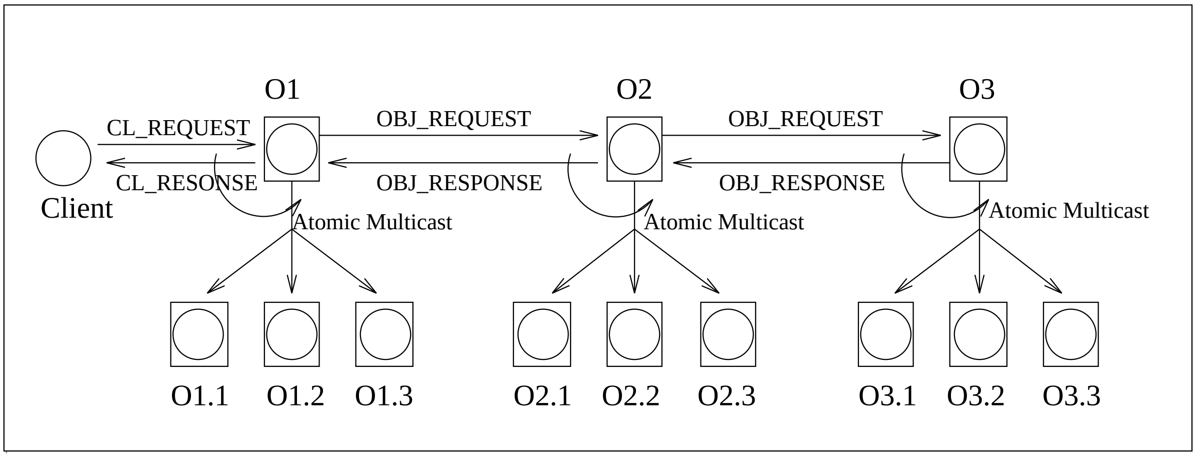

Figure 2: Replica Arrangement in Hot Replication Scheme.

4 The Algorithm for Hot Replication

In the *Hot Replication* scheme we arrange replicas (copies) of the object as *primary replica (or primary copy)* and *secondary replicas (or secondary copies).* By the *Replicated Object Set* we mean all the replicas of the object including the primary. *Secondary Object Set (Secondary Replica Set)* denotes all the copies of the object that act as secondary at a particular time. As far as the algorithm is concerned all the objects in the secondary object set are equivalent. The algorithm assumes that it has a way of knowing all the members of the replicated object set in the system. In Phoenix, the Fault Tolerant Naming Service (FTNS) provides this facility. Note that in the algorithm, when we say that the object (primary or secondary) performs a certain action, it means that the FTO layer (see figure 1) associated with that object performs that action. The object by itself does not participate in the algorithm. All it does is to service the request from the clients and invoke the services of the other objects (if necessary) through remote method invocation (perform a nested object invocation). The FTO Layer traps these messages and implements the logic of the algorithm to provide the fault tolerance. Thus the fault tolerance provided by the *hot replication* algorithm is fully application transparent. Figure 2 shows the arrangement of the objects in the hot replication scheme.

In the figure objects {O1, O1.1, O1.2, O1.3} form a replicated object set. Objects {O1.1, O1.2, O1.3} form a secondary object set for current primary object $O1$. Similarly {O2, O2.1, O2.2, O2.3} and {O3, O3.1, O3.2, O3.3} are replicated object sets. Here the degree of replication is four. Thus this configuration can tolerate three crash failures. In the figure, the circles show the application (user) objects (servers). The enclosing squares indicate that the messages to (and from) the object are filtered (passed) through the Phoenix system.

4.1 Overview of the Algorithm

The hot replication algorithm assumes a client-server setup in which client communicates with the server in a blocking *request-reply* sequence. The server objects are replicated. For obtaining any service the client (this is a non-replicated client) makes an invocation to a replicated object. In the system this client request can be uniquely identifiable, say by a transaction identifier[12]. The algorithm attaches a sequence number to every message it handles. We assume that the algorithm for generating the sequence number is deterministic. Thus in a given execution, the messages generated by the primary object and the messages generated by the objects in the secondary object set will get the same sequence number. The algorithm maintains a buffer for buffering the messages at the secondary.

We will briefly explain how an object invocation is carried out by the Phoenix FTO layer on behalf of an application object. The algorithm has four main logical modules namely: *Send(), Receive(), Check_in_buffer()* and *Become_primary().* The FTO layer uses these modules appropriately. When the application object generates OBJ_REQUEST message, the FTO layer invokes the *Send()* routine to send the message to the destination and to replicas. It then invokes the *Receive()* routine which *waits* for the corresponding OBJ_RESPONSE message. During this time if the *failure detection service* informs it that the destination has failed, it re-transmits the OBJ_REQUEST message again using the *Send()* routine. In the following we will briefly explain the modules used in the algorithm. Figure 3 and figure 4 show the high level logic of the modules.

4.1.1 Send() Module

This module is invoked, when the object sends a message. If the sending object is primary, then the message is reliably multicast to the destination object (primary of the destination object set) and to the secondary replica set of the primary. When the sending node is the secondary we enter the message in the buffer using *check_in_buffer()* routine. The details of this routine is given below. Note that if the system has to be fault tolerant (with respect to an object) then all the objects it interacts with (directly or indirectly) should be replicated.

4.1.2 Receive() Module

This module is invoked when there is a message to be received. If the receiving object is the primary, then the message is first reliably multicast to the secondary replica set. Then it is delivered to the object (the primary) for processing. If the receiving object is secondary, two cases arise. If the message type is CL_REQUEST or OBJ_RESPONSE, then the secondary object actually needs this message for processing. Thus it is delivered to the object for processing. Otherwise, the message type is CL_RESPONSE or OBJ_REQUEST. These messages are multicast (forwarded) from the primary to the secondary so that the secondary knows that

```
/*The following routines are executed during
the failure free execution */

Send()
{
  if (object = primary)
      multicast_msg(secondary_replica_set, destination);
  else
      check_in_buffer(msg, from_secondary);
}

Receive()
{
  if (object = primary){
      multicast_msg(secondary_replica_set);
      deliver_msg(object);
      }
  else {
      if (msg_type = CL_REQUEST or OBJ_RESPONSE)
          deliver_msg(object);
      else
          check_in_buffer(msg,from_primary);
      }
}

Check_in_buffer(msg,from)
{
  if (there is a buff_msg in buffer
      AND msg.SeqNo = buff_msg.SeqNo)
      delete(buff_msg);
  else
      enter_the_msg_in_the_buffer(from);
}
```

Figure 3: The Hot Replication Algorithm.

the primary has sent those messages. Thus, these messages are entered in the secondary buffer using *check_in_buffer()* routine.

4.1.3 Check_in_buffer() Module

This module is called by the secondary whenever it has to handle a message that cannot be delivered to (i.e., that is not intended to be received by) the object for processing. Note that the outgoing messages from the secondary and CL_RESPONSE, OBJ_REQUEST messages forwarded from the primary fall into this category. This routine has two arguments. The first one is the message (msg) and the second is the originator of the message. The originator can be the *primary* (i.e., message is forwarded by the primary multicast) or the *secondary* (i.e., secondary object wants to send a message out, may be OBJ_REQUEST or CL_RESPONSE). This routine checks the buffer to see that whether a message of same sequence number exists in the buffer. If there is such a message, it deletes that message. Note that the existence of such a message in the buffer indicates that the primary or the secondary had already sent

```
/* The following routine is executed by one of the
   secondaries after the primary has crashed */

Become_primary()
{
  process all the messages in the receive queue
  /*i.e, messages that are to be received */
  using the hot replication algorithm
  as given in figure 4.

  if (buffer is EMPTY){
      make the secondary as primary;
      return;
      }
  else {
      multicast_msg(secondary_replica_set, destination);
      make the secondary as primary;
      }
}
```

Figure 4: The Become_Primary Algorithm.

out that message and the other is just catching up. If there is no such message, then the message is entered in the buffer. While entering the message in the buffer, its originator (primary or secondary) is also stored along with the message (this information is used by the Become_Primary() module).

4.1.4 Become_Primary_Algorithm

Figure 4 shows the Become_Primary_Algorithm. This algorithm transforms a secondary object to a primary object. This module is executed by the Phoenix FTO Layer when it learns that the current primary has failed (the Health Checking Service of Phoenix informs the failure of the primary to the FTO Layer). The FTO Layer determines which one of the secondary objects in the secondary replica set should become primary. When this module is invoked on a secondary object it performs the following actions. First it processes (using the algorithm given in figure 4) all the messages in the receive (input) queue of the secondary. Note that these messages are intended to be received by the secondary (these messages must be processed by the secondary so that the secondary state catches up the primary state). When all these messages are processed (delivered), following two cases arise: a) *The output buffer is empty.* This means that the secondary state is synchronized with the primary state. Thus it can be made as the primary. b) The output buffer is not empty (note that there can be only one message in the output buffer, this will be examined later) means that the primary had not sent that message to the destination. So we multicast it to the destination and the secondary replica set. Now the buffer becomes empty and the secondary is made as primary.

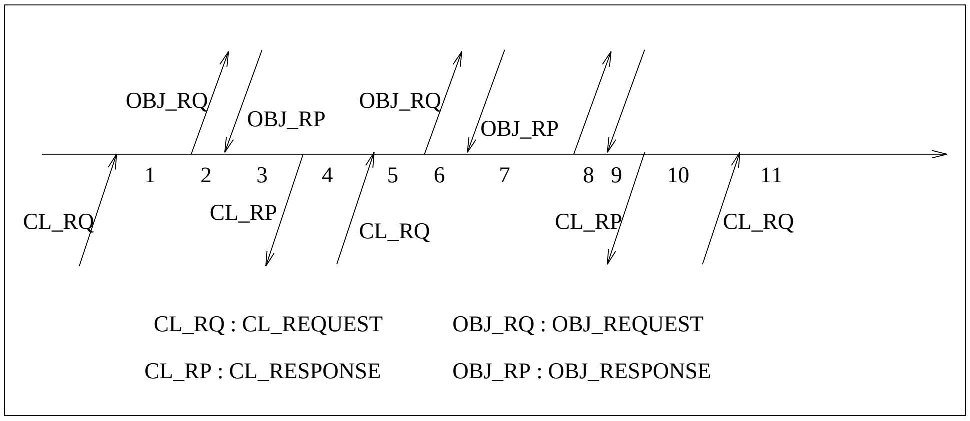

Figure 5: An example execution of the primary object.

4.2 Proof of Correctness

In this subsection, we informally argue about the correctness of the algorithm. We will consider the following cases:

Failure Free case: Note that only the primary interacts with the client and the other objects (for the nested invocations). Thus the problem of duplicate (same images of the top level invocation) executions in the nested object invocations does not arise in this case (the problem is discussed in section 2). The primary atomically multicasts all the messages it receives and the messages it sends to the secondary replica set. Since we assumed deterministic execution of the objects, and since the secondaries receive the same requests (message) as the primary (after executing those requests), the secondaries (secondary states) are always synchronized with the primary (primary state), although secondaries may be lagging or leading the primary in time.

One might think that it is not necessary to multicast CL_RESPONSE and OBJ_REQUEST messages to the secondary replica set, since these will be generated anyway by the secondary object during their normal execution. However, these multicasts are necessary to tolerate the failure of the primary. Figure 5 shows the execution of the primary. For our discussion, the state of the primary (object) changes after each send or receipt of a message. The primary can fail at any one of the numbered segments. By multicasting each message received or sent, the primary informs the secondary, exactly at what segment the primary is currently executing.

Failure Case: The *failure of any secondary* can be handled easily. In this case we do nothing, since the primary is still active and is servicing the client requests. In fact, it is not even necessary to detect the secondary failures (also see the discussion in section 4.4). We need to monitor only the liveliness of the primary. Thus the secondary failure does not affect the system. This is one of the major advantages of this scheme.

The *failure of the primary* is handled differently. The failure of the primary is detected by the Phoenix health checking service. It informs Phoenix FTO Layer about the primary failure. The FTO Layer chooses one of the secondaries to serve as the next primary (the choice may be made by looking at the size of the buffer at the secondaries). Let the secondary object $O1.1$ be selected as the primary. The following two cases arise: a) Secondary was slower than the primary, b) Secondary was faster (or of the same speed) than primary.

Secondary was slower than primary means that it is ($O1.1$) still executing some requests already serviced by the primary $O1$. So we simply let it continue the current execution and catch up to the primary state (using the algorithm given in figure 4). At the end of this processing, the buffer of the $O1.1$ may be empty or may contain only one message. The buffer of $O1.1$ **empty** means that for each request received by the primary, the corresponding response had been sent back to the client. Note that we are making use of the synchronous nature of the object interactions. Each request will have a single response associated with it. So now the state of $O1.1$ is the same as the state of $O1$ before it failed (strictly, O1.1 is in the same segment as was the primary just before the primary failed. See figure 5). Thus $O1.1$ can now act as the primary and can start servicing the fresh requests. The buffer of $O1.1$ **not empty** means that the primary $O1$ has not sent out a message corresponding to the last request being serviced by $O1$. This message may be a response to the client of an object request (for a nested invocation). Thus the message in the buffer must be a CL_RESPONSE or OBJ_REQUEST. The message is multicast to the destination (called object or the client) and the secondary replica set. Note that this is what the primary would have done had it not crashed. Now the object $O1.1$ is ready to act as the primary.

Secondary was faster than primary, in our case, implies that the secondary can generate at most one outgoing message before the primary does (and then the secondary has to block for the corresponding response message). This is because the response for this outgoing message has to be forwarded by the primary. Thus this case is similar to the *buffer of* $O1.1$ *is not empty* situation explained above. The same argument holds.

4.3 An Example

Now we will illustrate the hot replication algorithm with the help of an example. Consider figure 6 for this example. The degree of replication is three. So this can tolerate two crash failures. Initially the client sends a request (CL_RQ) to the object $O1$. To service this request, the object $O1$ needs to invoke the service of object $O2$. During the normal execution, the CL_RQ is multicast to $\{O1.1, O1.2\}$ and CL_RQ is delivered to $O1$. When OBJ_RQ is generated by $O1$ it is multicast to $\{O1.1, O1.2, O2\}$. Before this message is delivered to object $O2$ it is multicast to $\{O2.1, O2.2\}$. When $O2$ generates message OBJ_RP it is multicast to $\{O2.1, O2.2, O1\}$. When OBJ_RP is received by $O1$ it is multicast to $\{O1.1, O1.2\}$ before it is delivered to $O1$. Now

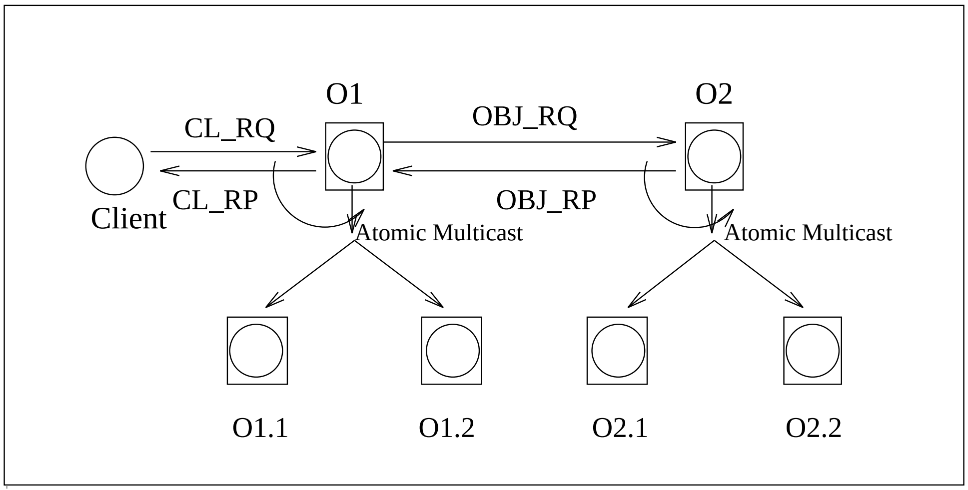

Figure 6: An example run for the Hot Replication Scheme.

consider that the primary (O1) has failed after it receives OBJ_RP and before it send CL_RP to the client. Let the FTO Layer chooses object $O1.2$ to become primary. If $O1.2$ was slower than the primary (its output buffer is empty) it will execute as secondary until the CL_RP message is generated. Then it acts as primary and multicasts CL_RP message. If $O1.2$ was faster than the primary, its output buffer will have CL_RP message. So it will multicast this message and then act as the primary.

4.4 Discussion

In this section we will make some comments about the *hot replication* algorithm. The first one is about the assumption on the communication primitives (these communication primitives will be provided by the Phoenix FTCS layer). We assumed two communication primitives: a) reliable point-to-point communication (used by the clients to request the object services), and b) atomic multicast communication (used for other communication required by the protocol). The atomic multicast primitive needs some elaboration when used with *hot replication* algorithm. Note that we gave two parameters for the *multicast_msg()* routine. The first one is a group (secondary replica set) and the second one is the destination of the message. The *multicast_msg(replica_set, destination)* has the following behavior: a) If the sender does not fail, then all the functioning members of the the secondary replica set and the destination receive the message, b) If the sender fails, then, either all the functioning members of the secondary replica set and the destination receive the message or none of the members of the secondary replica set and the destination receive the message.

The second comment we want to make is about the failure handling. Once the current primary fails, the new primary can either be elected using *leader election* algorithm or be selected by some pre-determined criteria. The failure of the secondary can be ignored as far as the correctness of the algorithm is concerned. However for performance reasons, we need to exclude the failed secondary from the replica set (so that the multicast response will be faster). Also if it is necessary to maintain a certain minimal reliability level, then we need to add new replicas into the replica set.

The third comment is about the handling of duplicate messages. Note that because of failures and retransmissions, an object may receive duplicate requests. The FTO layer should be able to detect these duplicate requests. The duplicate requests should not be delivered to the object. However it is necessary to send the corresponding response to these requests. Thus the FTO layer has to store the *response messages* (after multicasting the messages to the secondaries and the destination). These stored messages can be discarded after a suitable *time-out* period.

The final comment is about the performance of the *hot replication* algorithm. To the best of our knowledge, only two previous works[11, 12] deal with the problem of fault tolerance of the objects supporting nested object invocations (arbitrary level of nesting) using replication mechanism. (We exclude the checkpointing schemes here, because in situations where we can not use checkpointing, maybe due to the recovery time involved, we have to use replication based schemes.) The algorithm in [12] can not be meaningfully compared to our algorithm, since it uses a specific broadcast based network (otherwise it requires an ordered reliable broadcast implementation, which will be quite expensive). The algorithm [11] can be compared to *hot replication* scheme, as given below.

The main difference between our scheme and the scheme proposed by Yap et.al[11] is that we arrange all the secondary copies as a single group. The primary reliably multicasts the requests and results it receives to the secondaries (hot replicas). In [11] the secondaries are arranged as a chain, replica $i+1$ acts as a backup for replica i. In this case the object response time is linearly proportional to the number of replicas. Also the communication protocol and replication protocol are tightly coupled. In our scheme, the communication and replication protocols are separated (and independent). This can take advantage of better communication protocols or underlying network topology for improving the performance without changing the replication protocol.

In the *hot replication* scheme, if all the objects execute approximately in the same speed, then in the case of the primary failure, the secondary is immediately available for service (in fact, after time T_f, the failure detection time). If the secondary is faster, the same argument holds. If the secondary is slower, then we need (*catch up* + T_f) time to recover from a primary failure. Note that in practice the *catch up* time can be very much smaller than the recovery time associated with the checkpointing and rollback scheme. The *failure free overhead* is dictated by the efficiency of the multicast protocol used. Here, the modularity (separation of replication and communication algorithms) of our algorithm helps. We can use a more efficient multicast algorithm (or can make use of available special hardware features, like the ethernet broadcast) if available, without modifying the other parts of the system and can get a better performance.

5 Conclusion

In this paper we presented a new algorithm for providing transparent fault tolerance support for distributed objects in distributed systems. We use a *Hot Replication* scheme for providing fault tolerance. If there are k replicas (copies) of the objects in the object replica set, then the system can tolerate $k-1$ crash failures. The chief advantages of the scheme are : a) No action is needed in the case of failure of a secondary replica, b) The time to recover from a primary failure is minimal (this time is needed only if the secondary execution speed is slower compared to the primary execution speed), and c) Separation of the replication protocol and the reliable communication protocol. To recover from a primary failure the designated secondary has to just catch up to the primary state by processing outstanding messages, if any. This is in contrast with the checkpointing and rollback recovery scheme, where the recovery time can be substantial. However, in this scheme, the computing resources are used for all the objects in the object replica set. But assuming that, in general, the computing resources are lightly loaded this may not be considered a drawback. Note that even in the checkpointing scheme, for an actual implementation of the assumed stable storage abstraction, we may have to pay this price. As a part of our ongoing Phoenix project, we are implementing this algorithm on the SPRING[15, 16] Object Oriented Operating System.

References

[1] R. E. Strom and S. A. Yemini, "Optimistic recovery in distributed systems," *ACM Trans. Comp. Syst.*, vol. 3, no. 3, pp. 204–226, August 1985.

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

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

[4] 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.

[5] P. Jalote, "Fault tolerant processes," *Distributed Computing*, pp. 187–195, 1989.

[6] D. B. Johnson and W. Zwaenepoel, "Recovery in distributed systems using optimistic message logging and checkpointing," *Journal of Algorithms*, vol. 11, pp. 462–491, September 1990.

[7] G. Beedubail et al., "An algorithm for supporting fault tolerant objects in distributed object oriented operating systems," In *Proc. of International Workshop on Object-Orientation in Operating Systems*, August 1995.

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

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

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

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

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

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