An Algorithm for Supporting Fault Tolerant Objects in Distributed Object Oriented Operating Systems

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_95-019)
April 1995[1].

Abstract

This paper presents a new algorithm for supporting fault tolerant objects in distributed object oriented systems. The fault tolerance provided by the algorithm is fully user transparent. The algorithm uses checkpointing and message logging scheme. However the novelty of this scheme is in identifying the checkpointing instances such that the checkpointing time will not affect the regular response time for the object requests. It also results in storing the minimum amount of object state(object address space). A simple message logging scheme that pairs the logging of response message and the next request message reduces the message logging time by half on an average compared to other similar logging schemes. The scheme exploits the general features and concepts associated with the notion of the objects and object interactions to its advantage.

1 Introduction

Object oriented distributed operating systems are getting mainstream acceptance. They are maturing from research ideas to research prototypes[1, 2, 3] and soon will appear as commercial systems. When these systems are regularly used for critical applications, the reliability of the object services becomes a crucial issue. However system developers of these systems will neither have the time nor expertise to develop fault tolerant object services. It is thus imperative that the operating system should provide support for transparent fault tolerance for the objects that are executing on the operating system. In this paper we will present a novel scheme for providing such user transparent fault tolerance for objects executing on a distributed object oriented operating system.

Various schemes for providing fault tolerance for process/services in a distributed systems do exist in literature[4, 5, 6, 7, 8, 9]. Similar to some of those schemes, our scheme uses checkpointing and message logging. However the novelty of our scheme is that, it uses the general

[1]This Paper was revised in July 1995. The revised version appears in the Proceedings of the Fourth International Workshop on Object Orientation in Operating Systems (IWOOOS'95), Lund, Sweden, August 1995, pp 142-148.

characteristics of the objects (concept of object) to its advantage. Because of this, in our scheme the size of the object image (object state or object address space) that is checkpointed (and stored) is minimal. Also, the checkpointing time does not affect the response time for regular object invocation. Our message logging scheme reduces the logging time by half on an average compared to the similar logging schemes. During recovery, only the failed object need to roll back.

This work is done in the context of 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 failure detection service). The 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 the 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 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 the future work.

2 Related Work

In the literature, the subject of fault tolerance for distributed system is primarily addressed for process based systems with asynchronous message passing[5, 7, 10, 11, 12, 13]. A few systems also consider synchronous message passing like remote procedure calls[6, 14]. Although there exists a duality between object oriented systems and process (and messages) based systems[15, 16], not much work has been done in exploiting the structure and properties of the object oriented systems in providing fault tolerance[15].

Techniques for providing fault tolerance can be basically classified into two categories: a) *checkpointing and rollback recovery* and b) *process replication (or modular redundancy).* In the first approach, the algorithms presented fall into two classes: *independent checkpointing* and *consistent checkpointing.*

In the case of independent checkpointing, each process checkpoints its state independently. Since checkpointing is independent, the checkpoints of different processes may not define a *consistent state* (a *consistent global state* is one, in which messages are recorded as received only if they are also recorded as sent[17]). If the recorded state is not consistent, then while recovering from a crash this may cause rollback of other processes (cascaded rollbacks)[18]. Simple independent checkpointing schemes suffer with this problem. This problem can be solved by logging the messages in conjunction with independent checkpointing. This approach is generally known as message logging. In *pessimistic message logging* scheme[4, 5, 7] each message is synchronously logged on *stable storage* as it is received. When a process fails it can recover locally (without communicating with the other processes) by re-processing the messages from the stable storage log. Since this leads to high failure free overhead (synchronously logging messages on stable storage is expensive) *optimistic message logging* schemes[19, 8, 13, 9] were proposed. In this approach messages are logged asynchronously (many messages may be grouped together) on stable storage. When failure occurs, processes coordinate to determine an optimal consistent set of checkpoints and recover from that point. This may require multiple processes rollback to their earlier state even though they did not fail. Note that this does not happen in the case of pessimistic message logging case. The recovery is only a local affair.

In the consistent checkpointing schemes[18, 19] all processes coordinate the checkpointing activity such that the set of checkpoints form a consistent global state[17]. No application related processing is carried out during the execution of the checkpointing algorithm. In the algorithm proposed by Koo and Toueg[18], consistent checkpointing is taken by a two-phase message exchange. This algorithm saves two kinds of checkpoints in the stable storage: *tentative* and *permanent*. When a process P_i takes a tentative checkpoint, it forces all the processes to take a checkpoint that have sent messages to P_i, since the last checkpoint of P_i. Each dependent process executes the same operation in a recursive way. During this time P_i blocks for a response. When P_i gets the response, it changes the tentative checkpoint to permanent checkpoint and sends commit messages to the dependents. When process P_k rolls back due to a failure, it forces the rollback of all processes which has received any messages from P_k. Koo and Toueg show that this algorithm is optimal in the sense that, it forces a minimum number of processes to take a checkpoint and to rollback. However this algorithm makes an assumption that concurrent invocation of algorithm is not in effect. The algorithm presented by Leu and Bhargava[19] allow higher concurrency among the invocation of checkpoint and rollback algorithm.

Some other interesting variations of the above checkpointing schemes are found in[4, 20]. In the algorithm presented by Birman et.al.[4] for implementing *k-resilient* objects, an independent checkpointing scheme is used in conjunction with the *retained results*. They propose a *coordinator-cohort* scheme in which the coordinator checkpoints its state at the cohorts independently and the results of the call (along with a unique activity id) are also stored at the cohorts (this is basically a variation of pessimistic message logging). During the recovery, these retained results is available for the failed processes (the call is not re-executed). The *sender based message logging* scheme was proposed by Johnson and Zwaenpoel[20]. This algorithm can tolerate only one fault in the system (one node crash). But the message is logged in the volatile storage (main memory), and thus the message logging overhead is substantially lower.

Modular redundancy (or process replication) takes a different approach for fault tolerance[6, 11, 14, 21, 22]. This approach generally assumes a client-server based distributed system. The server (and maybe the clients too) is replicated. Each replica of the server executes the client request concurrently and sends a reply. The client discards the duplicate replies. In [22] Fred Schneider presents a detailed model for this approach, which he calls a *state machine approach* for implementing fault tolerant services. He discusses the agreement and the order requirements of the application messages for this scheme. Eric Cooper describes[6] replicated procedure call (RPC) mechanism for constructing highly available distributed programs.Pankaj Jalote describes[14] a fault tolerant RPC provided by employing a combination of the modular redundancy approach and the primary standby approach. In [21] he also 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. Elnozahy describes the *Manetho* system[11] which employs both replication and rollback recovery methods for providing transparent fault tolerance for distributed applications.

A checkpointing scheme that exploits the structure of object based system is given by Lin and Ahamad[15]. This algorithm uses a consistent checkpointing scheme similar to the algorithm given in[18]. However the algorithm given in[18] does not consider the process behavior while deciding the checkpoint and rollback dependencies. But in the algorithm given in[15], the checkpoint and rollback dependencies are derived considering the operational behavior of the invoked objects. The basic idea is classify the operations as *lookup* or *modify* operations. They assume that the type of operations can be identified in most cases from its source code. Thus by utilizing the semantics of the object operations and invocations the number of objects involved in the checkpointing and rollback are minimized.

Our algorithm also uses the behavior of the objects and operations of the objects but quite differently than that used in[15]. We do not assume the semantics of the object operations for two reasons. First, it is quite quite difficult to get the semantics of the object operations by parsing the source code (unless we use a special language in which object states are somehow tagged). Second, we wanted to provide user transparent fault tolerance for third party objects for which the source code is not available.

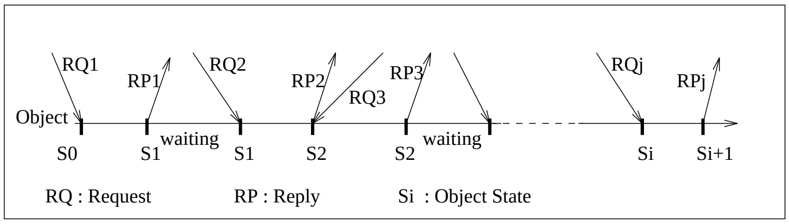

Figure 1: A typical execution segment of an object in its lifetime.

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 the services to the user. The objects interact with each other and with the outside world through well defined interfaces. The programming model used in this system is the object oriented paradigm. The object interfaces are defined in such a system using some standard interface definition language (IDL), though this fact is not necessary (or assumed) for our algorithm. Systems such as SPRING[2, 3] provide such features. The objects in the system are active (i.e. the object is always executing for providing services).

3.1 Objects

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. The servicing of each request may change the internal state of the object. If the object initially starts with a (initial) state of S_0, the successive request-reply sequence may take the object state from S_0 through the sequence $S_1, S_2 \cdots S_i, S_{i+1}, \cdots$. Note that all the request-reply sequences may not change the object state. The figure 1 illustrates this point. In the figure, the object starts its execution with an initial state S_0. The request-reply sequence $(RQ_1 - RP_1)$ takes the state of the object to S_1. After the reply (RP_1) there is no immediate request for the object, thus it waits for the next request (no state change takes place during this time). So when request RQ_2 arrives the object state is still S_1. When reply RP_2 is sent the object state is changed to S_2. Request RQ_3 is serviced immediately after sending $RP2$ (there is no waiting). Note that the servicing of the request $RQ3$ did not change the object state (it might have been just a query/read request). The similar pattern continues during the lifetime

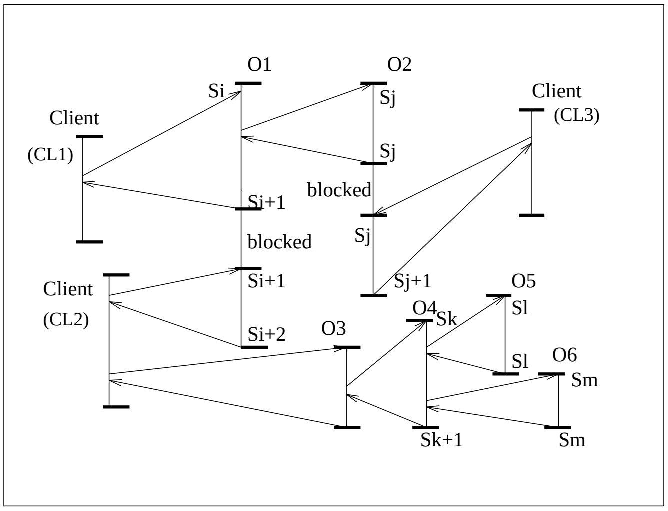

Figure 2: An example of nested object invocation.

of the object.

For serving a request the object may invoke the services 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). Note that all objects in the system behave similarly. Figure 2 shows an example of nested object invocation. Note that in this figure CL_1, CL_2 and CL_3 are clients (short living objects) and $O_1, \cdots, O_6$ are the servers (long living objects). Also O_1 behaves as a client for O_2 for some invocations (similarly other objects can invoke services of different objects). We assume here that the execution time of each invocation is short (may be a short transaction). This is true in most client-server based systems.

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.

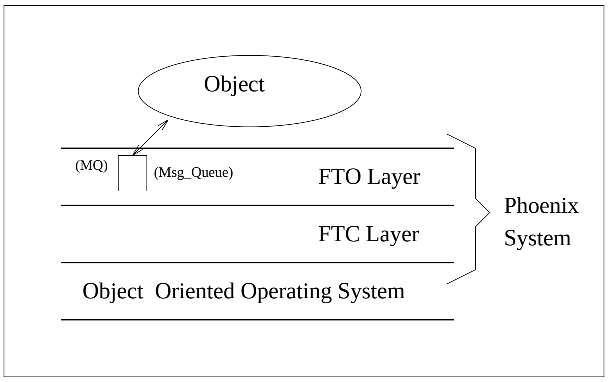

Figure 3: The System Model

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. In the Phoenix system these services are provided by the various Phoenix components. Figure 3 shows the assumed model of the system. We also assume that a stable storage service is available in the system. We assume that the nodes in the system can fail by crashing (i.e. fail stop assumption).

4 The Algorithm

The main motivation (idea) for our algorithm comes from observing the operation of a typical object. Any typical object operates in a sequence such as the following:

- *wait_for_request*
- *do_computation (and_zero_or_more_other_object_invocations)*
- *reply_to_the_request*

Note that when object sends a request to other objects, it blocks until it gets a reply (synchronous communication).

The main idea of the algorithm is to take checkpoints when the object is waiting for the client (or other object) requests. That is, when the object is not serving any client/object request. Thus the response time of the object is not affected. Since most of the time, the object will be in *waiting-for-the-client-request* state, we can take a check point at this time (basically when the request queue is empty). An assumption here is that, the average checkpointing time is small compared to the mean inter arrival time of the requests. Also note that when object is

```
FTO_Algorithm ()
{
  Read_the_msg_from_MQ ();
  switch (MSG_TYPE){
  case CL_REQUEST:
            LOG (log_msg = CL_REQUEST_msg);
            CheckPoint (NOTEMPTY);
            msg_counter++;
            break;
  case CL_RESPONSE:
            if(MQ is not empty){
             LOG_SEND(log_msg =
                      [CL_RESPONSE_msg + next CL_REQUEST_msg],
                      send_msg = CL_RESPONSE_msg);
             CheckPoint (NOTEMPTY);
             Deliver (CL_REQUEST_msg);
             msg_counter++;
             }
            else{
             LOG_SEND (log_msg = [CL_RESPONSE_msg],
                      send_msg = CL_RESPONSE_msg);
             msg_counter++;
             CheckPoint (EMPTY);
             }
            break;
  case OBJ_REQUEST, OBJ_RESPONSE:
            Deliver (msg); break;
  }/* switch () */
} /*FTO_algorithm () */

CheckPoint (status)
{
  switch (status){
  case EMPTY :
             if( msg_counter > MSG_LIMIT){
              check_point_the_object_state_in_SS (); msg_counter = 0;
              }
             break;
  case NOTEMPTY :
             if( msg_counter > MSG_LIMIT){
              if(num_tries > TRY_LIMIT){
               check_point_the_object_state_in_SS ();
               msg_counter = 0;
               num_tries = 0;
               }
              else num_tries++;
              }
             break;
  }/* switch () */
}/*Checkpoint ()*/
```

Figure 4: The Checkpointing and Message Logging Algorithm.

waiting for a client request, it is not executing, thus the memory image of the object (the object state that is to be checkpointed/stored) has the minimum size. This is because at this time the memory image will not have any temporary variables (stack variables). It will have only the necessary object state (object data) and object procedure code. In between checkpoints we will log the CL_REQUEST (CL_RESPONSE) message arriving to (departing from) the object. Note that we are logging only two messages for an object invocation (i.e., the Request and Response for the invocation). Any other messages arising out of this request (the requests/replies for any other invocation from this object to other objects), are not logged in the context of this object.

4.1 The FTO Algorithm

The figure 4 shows the checkpointing and message logging algorithm (FTO_Algorithm). This algorithm is executed by the Phoenix FTO Layer in the context of every fault tolerant object in the system (also refer to figure 3).

The routines *LOG_SEND (log_msg, send_msg)* and *LOG (log_msg)* are the services provided by the system. In our case fault tolerant communication (FTC) layer of the Phoenix provides this service. The LOG_SEND () routine will reliably log the log_msg in the stable storage and will send the send_msg to the destination. Both of this happens atomically. The LOG () routine will reliably log the message in the stable storage.

The *CheckPoint (status)* routine checkpoints the state of the object. This routine is called when the object is not servicing any request. There are two pre-defined input parameters for this algorithm: MSG_LIMIT and TRY_LIMIT. This routine is called every time when a response (or response and next request) message is logged. If MSG_LIMIT messages had been logged already (means that MSG_LIMIT requests have been serviced since the last request), then if the request queue empty, a checkpoint is taken. If the request queue is not empty then we have some pending requests. Since we do not want to effect the normal response time by taking checkpoint, we wait for another TRY_LIMIT service requests, to see whether the request queue becomes empty. If so, then we will take the checkpoint at that time. If not then we are forced to take the checkpoint (when there are requests in the queue). Otherwise we will loose lot of computation (i.e. the recovery time will be long). The values for the MSG_LIMIT and TRY_LIMIT can be decided based on the average processing time for each request and average checkpoint time (basically optimizing the overhead for fault tolerance). Note that we are not taking the checkpoints based on a real time interval (say, for every one hour) since the object may not be active uniformly over the whole time interval (i.e. the object state change is not continuous with respect to time).

4.2 The Recovery Algorithm

The recovery algorithm is initiated by the Phoenix FTO layer when the Phoenix Health Checker informs it that some (fault tolerant) object in the system has failed (maybe due to some node crash). The recovery algorithm affects only the failed object. All other objects in the system are unaware of the fact that some objects are recovering from a failure. Figure 5 shows the recovery algorithm.

4.3 Correctness and Explanation

When an object fails, it is restored to its last checkpoint. Then all the messages that were delivered to that object are replayed from the stable storage. All messages that were send

```
FTO_recovery_algorithm ()
{
  restore_the_state_of_the_object_from_stable_storage ();
  Recovering = YES;
  while (Recovering == YES){
      read_next_logged_msg_for_this_obj_from_SS (); /* CL_REQUESTmsg */
      if(no_more_logged_msgs){
        Recovering = NO;
        start_normal_execution ();
        }
      deliver_msg_to_object ();
      for_all (outgoing_request_msgs_generated_by_this_object){
        ignore_it ();
        use_its_msg_id_to_get_the_corresponding_reply_msg_fromSS ();
        if (reply_msg_found) deliver_it_to_object ();
        else {
          Recovering = NO;
          start_normal_execution ();
          }
      }/* for_all */
      for (the_reply_generated_by_this_object){
        check_whether_the_reply_msg_is_in_SS ();
        if (YES) ignore_the_reply ();
        /* the reply msg had been delivered before */
        else LOG_SEND (log_msg = reply_msg; send_msg = reply_msg);
      }/* for */
  } /* while () */
}
```

Figure 5: The Recovery Algorithm.

from the failed object, and that already exist in the log are ignored. The relative order of messages is preserved because of the assumption that all calls to objects are blocking (i.e., a client object cannot have made two concurrent calls to two different objects). Since the objects are assumed to be deterministic, at the end of the recovery protocol object will be in the same state at which it was there before the crash. The logging and reliable delivery of messages is handled by the fault tolerant communication (FTC) layer of Phoenix, which provides reliable and atomic delivery mechanism for the messages. Note that by logging CL_REPLY and the next CL_REQUEST messages together we reduced the message logging time (on an average) by half compared to other similar pessimistic logging schemes[5, 7]. Note that once we successfully take the latest state checkpoint we can delete the old state checkpoint and the logged messages.

We will explain the recovery algorithm using an example. Refer to the figure 6. For simplicity messages are numbered as $m_1, m_2, \cdots$ etc. Here we have two objects O_1 and O_2. The states of the objects are checkpointed at states S_i and S_j (as shown in the figure 6). During the normal run the request of the clients CL_1 and CL_2 and the associated object invocations as as shown in the figure 6. When the object O_1 is at the state F_1, it crashes. The health checker detects this crash and informs the Phoenix FTO layer. The FTO layer initiates the recovery protocol for object O_1. Note that in the stable storage (SS) we have checkpointed object O_1 at state S_i and object O_2 at state S_j. We also have logged messages m_1 through m_8. During the recovery action the following sequence of events take place. At the start of recovery the state of the object O_1 (state S_i) is restored (may be in some other node). Then message m_1 is replayed to O_1. Since m_2 is already in SS, it is not sent to client CL_1. Then m_3 is delivered to O_1. During

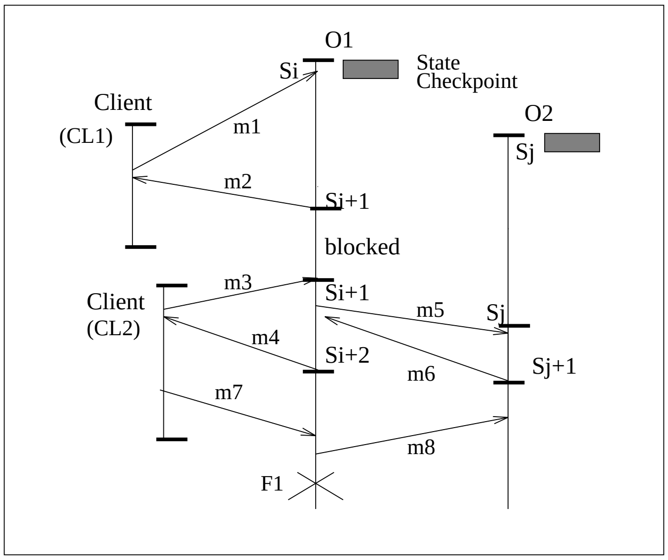

Figure 6: An example for Recovery Protocol.

the recovery execution the message m_5 is regenerated, but it is not sent to O_2. Instead the reply message m_6 is got from SS and delivered to O_1. Message m_4 is in stable storage and thus it is not sent to CL_2. The next request message m_7 is delivered to O_1 from the SS. Message m_8 is not sent to O_2 since it is in SS. (We assume that O_2 had logged it. Otherwise it will be resent.) Now there are no more messages logged for object O_1 in the SS. Thus the recovery of O_1 is complete and it starts its normal execution. Note that while O_1 is recovering, O_2 might have sent reply message for m_8 through FTC layer. Thus the FTC will deliver the reply for m_8 when O_1 is recovered to the state F_1.

4.4 Discussion

In this section we will discuss about the cost of our algorithm (qualitatively) compared to the other checkpointing algorithms[5, 7, 8, 11]. The favorable metric to our algorithm is the response time overhead introduced by the checkpointing. As discussed previously, average of this time will be negligible since in general the checkpoints are taken when the object is not servicing any request. This is generally true if the server is lightly loaded. Note that the server behaves like a single server queuing system. When the server is lightly loaded, the server will be in idle state most of the time. Our algorithm recognizes this fact in an application independent way. In other checkpointing algorithms, checkpoint interval is based on other parameters (in most cases it is the real time elapsed since the last checkpoint or a function of the number of messages received since the last checkpoint). So there is a high possibility that the checkpointing time will delay the object response time.

The performance of our algorithm can be still improved by the incorporation of the other techniques found in the literature, such as incremental checkpointing, concurrent checkpointing, or using the memory management hardware to implement the checkpointing. Using the application dependent knowledge for the checkpointing will reduce the cost of checkpointing substantially (we can checkpoint only the application data, no need to checkpoint the application code). But of course, we will loose user transparency here. The requirement of the stable storage and reliable message transmission is inherent in any of the checkpointing and message logging schemes. Thus we will not elaborate on these aspects here.

5 Conclusion

In this paper we present a new algorithm for providing transparent fault tolerance support for distributed objects on a distributed object oriented operating systems. The novelty of our scheme is in identifying the checkpointing instances, such that the checkpointing time will not affect the normal response time for the object invocation. This scheme also results in minimizing the object state that is to be stored during the checkpointing. This minimization results because, at the checkpointing time, the object will be in a blocked state (waiting for a client request), and thus the object image (object's address space) will not have any stack (temporary) variables. Our scheme for logging the messages between checkpoints is quite elegant and on an average may reduce the message logging overhead by half (compared to other similar logging schemes). Here we are using pessimistic (synchronous) message logging scheme, which maybe expensive compared to the optimistic (asynchronous) message logging scheme. However for the environment we are considering (client-server based environment), we believe that the synchronous logging scheme is the only solution. This is because, in these environments, client objects are short living entities. The interaction of the long living (server) objects with the client objects requires output commit actions[10] which requires a synchronous message logging. Thus we have to log all the object invocations synchronously. This is because, in general it is not possible to distinguish an object invocation either as an invocation from a client or as an invocation from (another server) object and we want to implement the FTO_Algorithm without knowing application semantics.

As a part of our ongoing Phoenix project, we will implement this algorithm on the SPRING object oriented distributed operating system. Future work on this algorithm will be in enhancing this algorithm to support fault tolerant objects that has multiple threads running concurrently. The problem is to find out, when the object is in a quiescent state (i.e. in a blocked state, waiting for client requests). This may not be the time when the request queue is empty, since multiple threads may be servicing the requests concurrently. Also when there are multiple threads, the remote object invocation may no longer be considered as a synchronous invocation (for the request-response message ordering). We need to address this issue as well. We still need to assume that the object behavior is deterministic. Otherwise (the only way we can think of) to make an object fault tolerant, we have to checkpoint the object state every time a change in the object state occurs. This may be too expensive solution. Currently we are also investigating how the object structure and properties can be exploited in supporting efficient replicated objects.

Acknowledgments

The authors would like to thank Dr. Nitin Vaidya for his helpful comments and suggestions. The comments from reviewers helped to improve the quality of the paper.

References

[1] R. Lea, C. Jacquemot, and E. Pillevesse, "Cool: System support for distributed programming," *Communications of the ACM*, vol. 36, no. 9, pp. 37–46, September 1993.

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

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

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

[5] 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.

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

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

[8] 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.

[9] 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.

[10] E. N. Elnozahy and W. Zwaenepoel, "Manetho: Transparent rollback-recovery with low overhead, limited rollback, and fast output commit," *IEEE Trans. Computers*, vol. 41, no. 5, , May 1992.

[11] E. N. Elnozahy and W. Zwaenepoel, "An integrated approach to fault tolerance," In *Management of Replicated Data, 1992 Workshop*, 1992.

[12] L. M. Silva and J. G. Silva, "Global checkpointing for distributed programs," In *Symp. Reliab. Distr. Systems*, pp. 155–162, October 1992.

[13] A. P. Sistla and J. L. Welch, "Efficient distributed recovery using message logging," In *Proc. ACM Symp. on Principles of Distributed Computing*, pp. 223–238, August 1989.

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

[15] L. Lin and M. Ahamad, "Checkpointing and rollback-recovery in distributed object based systems," In *FTCS-20*, pp. 97–104, 1990.

[16] L. V. Mancini and S. K. Srivastava, "Replication within atomic actions and conversations: A case study in fault tolerant duality," In *FTCS-19*, pp. 454–461, 1989.

[17] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states in distributed systems," *ACM Trans. Comp. Syst.*, vol. 3, no. 1, pp. 63–75, February 1985.

[18] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *IEEE Trans. Softw. Eng.*, vol. 13, no. 1, pp. 23–31, January 1987.

[19] P. Y. Leu and B. Bargava, "Concurrent robust checkpointing and rollback recovery in distributed systems," In *4th Int. Conf. on Data Engineering*, pp. 154–163, 1988.

[20] D. B. Johnson and W. Zwaenepoel, "Sender-based message logging," In *Digest of papers: The* 17^{th} *Int. Symp. Fault-Tolerant Comp.*, pp. 14–19, June 1987.

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

[22] 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.