Efficient Checkpointing Algorithm for Distributed Systems Implementing Reliable Communication Channels

Eugene Gendelman
Lubomir F. Bic
Michael B. Dillencourt

Department of Information and Computer Science
University of California
Irvine, CA 92717-3425 USA
Email:{egendelm, bic, dillenco}@ics.uci.edu

Abstract

This paper presents a new checkpointing algorithm for systems using reliable communication channels. The new algorithm requires O(n + m) communication messages, where n is the number of participating processes, and m is the number of late messages. The algorithm is non-blocking, requires minimal message logging, and has minimal stable storage requirements. This algorithm is also scalable, simple, transparent to the user, and facilitates fast recovery. By introducing sutable delay in the checkpointing process, the parameter m can be made small. We also describe a variant of the algorithm that requires only O(n) messages, at a cost of O(n) additional storage for each process.

This paper also presents an efficient coordination mechanism, called the Process Order. The Process Order mechanism can be used for grouping processes in arbitrary structures in order to solve various problems, including scalability, failure detection, and coordinator election. The Process Order mechanism groups the processes transparent to the user, and automatically adjusts to the changes in system topology.

1 Introduction

Distributed systems consisting of a network of workstations or personal computers are an attractive way to speed up large computations. These systems have a much higher performance-to-price ratio than large parallel computers, and they are also more widely available.

The computing nodes in a distributed system may fail. As some applications may require hours to execute, it is important to be able to continue computation in the presence of the node failure. Recovery from failures becomes more important for large systems, since the possibility of a node failure increases with the number of computing nodes.

Failure recovery may be achieved with a rollback-recovery mechanism. A rollback-recovery mechanism consists of three parts: checkpointing, fault detection, and failure recovery. During checkpointing the states of the participating processes are periodically saved on a stable storage. The saved process state is called a *checkpoint*. When a node failure occurs, the recovery mechanism uses saved checkpoints to recover the system to the consistent system state and continue execution from that state. The number of processes that have to be rolled back to the previous checkpoint varies, depending on the recovery algorithm. It may be necessary for one [9], some [8], or all [17] processes to roll back to the previous checkpoint.

It is useful for the checkpointing algorithm to take a checkpoint of each process so that the set of the local checkpoints represents a consistent system state, also called a *consistent system snapshot*. This facilitates fast recovery, since a consistent system snapshot does not have to be derived from uncoordinated local checkpoints. It also simplifies memory management, since as a new consistent snapshot is taken, the previous one can be discarded.

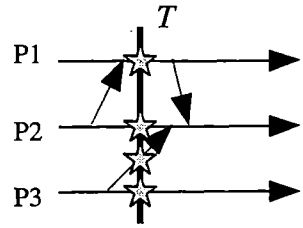


Figure 1

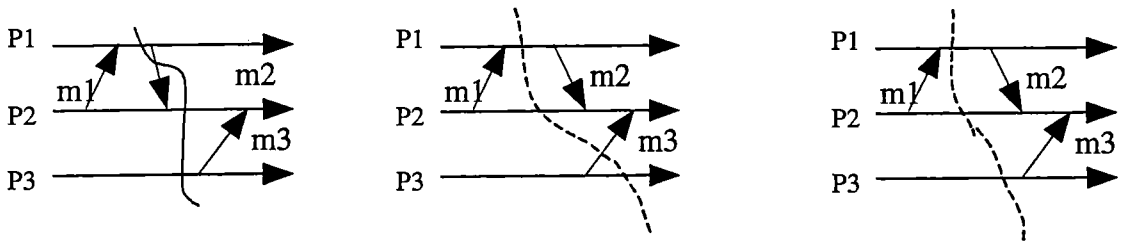
In this paper, we present a new algorithm for checkpointing in distributed systems with reliable communication channels. The algorithm is simple, efficient, and scalable. The organization of the paper is as follows. Section 2 discusses the notion of a consistent system state in the context of distributed systems with reliable communication channels. In Section 3, we present a list of useful properties for a checkpointing algorithm, and we briefly survey existing checkpointing algorithms. The new algorithm is presented in Section 4. In section 5, we present a simple mechanism for arranging processes in groups that does not require the exchange of additional messages; this is useful for forming the hierarchical structure used by our algorithm, and it has other uses as well.

2 Consistent System State

The system state at time T consists of the state of each participating process, plus every message in transit. This is illustrated in figure 1. The horizontal lines represent the progress of processes P1, P2, and P3 in time. Diagonal arrows represent messages exchanged by the processes, and the vertical line represents the time T. All the processes alive at T and all messages transferred at T belong to the system state at time T, as shown by stars.

The definition of a consistent system state varies, depending on the underlying communication channels. *Reliable* communication channels guarantee that every message that was sent will also be delivered to its destination. *Unreliable* communication channels dont provide such a guarantee. Intuitively, a consistent system state is one that may occur in a legal execution of a distributed computation [1]. For systems using unreliable communication channels *a consistent system state is one in which every message that has been received is also shown to have been sent in the state of the sender* [2]. In distributed systems using reliable communication channels a consistent system state also includes in-transit messages since they will always be delivered to their destination in any legal execution of the program. Hence, we add to the above definition that *a cosistent system state is one in which every message that has been sent is also shown to be received in the state of the receiver.*

Reliable communication channels are harder to provide than unreliable channels. The guaranteed message delivery may be provided by the hardware, by communication protocol, or by a user program. Supporting



a) received-not-sent b) sent-not-received c) consistent

Figure 2

reliable communication channels places a more stringent requirement on the checkpointing algorithm. If the checkpoints are taken on the level relying on the guaranteed message delivery, then the checkpointing algorithm must guarantee that all the messages that were sent were also received. The examples of such checkpoints include application-level rollback-recovery, or distributed systems relying on reliable communication channels.

Figure 2 shows the types of inconsistencies that checkpointing algorithms should avoid in order to take a consistent system snapshot. In these diagrams, checkpointing is represented by a dashed line. The checkpointing line is not strait because clocks of different computing nodes are not synchronized. Instead checkpointing is synchronized by message exchanges, and message transmission in asynchronous systems can take arbitrary time. When the system snapshot line crosses the process line, the checkpoint of the process is taken.

When a snapshot is taken as in figure 2a, message m2 will be registered as received by P2, but will not be registered as sent by P1. This is called a *received-not-sent* inconsistency. Figure 2b illustrates a *sent-not-received* inconsistency; m3 is registered as sent by P3, but it is not registered as received by P2. This type of inconsistency applies only to systems with reliable communication channels, since in systems that use unreliable channels message loss is allowed. The snapshot shown in figure 2c is an example of a consistent system snapshot.

3 Related Work

The goal of this project is to design a checkpointing algorithm with the following characteristics:

1. It should work correctly for systems that use reliable communication channels.
2. It should be non-blocking, meaning that the execution of the main computation is not blocked between steps of the checkpointing protocol. The reasons for it are discussed below.
3. The algorithm should incur as little overhead as possible during failure-free computation, where overhead accumulates from message logging, checkpoint coordination messages, and any other checkpoint-specific computations that need to be performed.
4. The algorithm should be scalable.
5. It should be general, i.e., it should not rely on any system-specific features.
6. It should be simple. This is important for the practical implementation of the algorithm, as well as for proving its correctness.

Depending on the system size, network delays, and the frequency with which checkpoints are taken, blocking may cause a system slowdown. In some cases the delay caused by blocking renders blocking algorithms unusable at all.

For example, as explained in [19], in the MESSENGERS system the execution thread is compiled to a set of the function blocks, so that the state of the process between the function blocks consists of the set of the local variables and the index of the next function to be executed. This allows to take checkpoints both independent of the computing platform, and also transparent to the user application. There are many advantages of such checkpointing when used in heterogeneous distributed system. To implement such checkpointing, the process state has to be saved between the function blocks. This means the process will not respond to the checkpointing message until it finishes executing the function block. Blocking algorithms will cause the whole system to halt, which might cause large delays.

There has been much research in designing checkpointing algorithms [2, 4-8, 10-17]. However, all known algorithms fail to satisfy one or more of the above requirements. Algorithms [2, 7, 10] are designed for systems that use unreliable communication channels. Algorithms presented in [7, 8, 11, 12, 17] are blocking.

Algorithms [14, 15] rely on the assumption that all the processor clocks are approximately synchronized, which limits the generality of these algorithms.

Some of these algorithms [2, 4, 5] require $O(n^2)$ communication messages, where n is the number of processors in the system. This makes them unnecessarily slow as the number of participating nodes grows. In Wojciks algorithm [13] each process has to log each message sent to other processes, which makes this algorithm inefficient.

Several non-blocking algorithms [6, 16] require less than $O(n^2)$ communication messages. The Kai Li algorithm [6] performs well on multicomputers. It requires $O(nlogn)$ messages for hypercube connected multicomputers and $O(n)$ for mesh connected multicomputers. However, this algorithm depends on the knowledge of the process interconnection topology, which is unusable in systems where the pattern by which processors are connected varies, as in systems constructed by interconnecting PCs or workstations. Furthermore, this algorithm requires communication channels to be FIFO.

The Silva algorithm [16] requires only $O(n)$ communication messages. However, it relies on the knowledge of fault detection latency, and message latency, which might be difficult to determine in case of the internet-based distributed system.

This paper describes a simple, scalable, non-blocking algorithm that requires $O(n + m)$ coordination messages, where m is the number of late messages (as explained below), which can be made small as described in section 4.2.3. The new algorithm requires minimal message logging. It does not require any system-specific information, and it works correctly in the presence of message reordering. The algorithm has minimal stable storage requirements, allows fast failure recovery, and provides simple garbage collection mechanism. These characteristics make this algorithm attractive for practical use.

4 Checkpointing Algorithm

The algorithm is presented incrementally. In section 4.1 we describe the algorithm designed for unreliable communication channels that was presented in [10], which forms the basis for the proposed algorithm. In section 4.2 we present an algorithm for checkpointing with reliable communication channels, using a central coordinator. Finaly, in section 4.3, we present the decentralized algorithm.

4.1 Algorithm For Unreliable Communication Channels

One distinguished checkpoint server acts as a checkpoint coordinator. Each process maintains one permanent checkpoint, belonging to the most recent consistent checkpoint. During each run of the protocol, each process takes a tentative checkpoint, which replaces the permanent one only if the protocol terminates successfully [7]. Each checkpoint is identified by a monotonically increasing Checkpoint Number (CN). Every application message is tagged with the CN of its sender, enabling the protocol to run in the presence of message re-ordering or loss [18].

The protocol proceeds as follows:

1. The coordinator starts a new consistent checkpoint by incrementing CN and sending an *Initiate* message that contains its CN to each process in the system.

2. Upon receiving an *Initiate* message, a process takes a tentative checkpoint, increments its local CN, and sends a *cpTaken* message to the coordinator.

If a process receives an application message with a CN greater than its own, it also takes a checkpoint before processing the message and then sends a *cpTaken* message to the coordinator.

3. When the coordinator has received a *cpTaken* message from all processes, and if all tentative checkpoints have been successful, it broadcasts a *Commit* message.

4. When a process receives a *Commit* message, it makes the tentative checkpoint permanent and discards the previous permanent checkpoint.

4.1.1 Message Loss

With the checkpointing algorithm described above, application messages can be lost. Consider the distributed system in figure 3 with four interconnected processes. Suppose that P0 is a coordinator and that it initiates checkpointing by broadcasting *Initiate* message (checkpointing messages cm1, cm2, cm3). P3 sends an application message am1 to P1. The message cm3 arrives at P3 *after* am1 is sent. P3 takes a checkpoint that registers that am1 was sent. The message cm1 arrives at P1 *before* am1. P1 takes a checkpoint that does not register am1 as a received message. If the system recovers from this snapshot, message am1 will be lost.

To resolve this problem, messages arriving at the node after checkpoint was taken, or *late messages*, have to be included into the system state. Since a checkpoint is not completed until all late messages arrived at their destination, the algorithm also needs to decide when the checkpoint is completed and could be committed.

4.2 Algorithm For Reliable Communication Channels

In the new algorithm, when a late message arrives at a node, a copy of a message is appended to the corresponding checkpoint. A late message is identified by the checkpoint number attached to it. If the messages CN is less than the receiver process CN, then the message has to be appended to the checkpoint identified by the message CN.

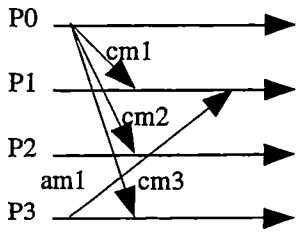


Figure 3

Another task of the algorithm is to determine when the checkpoint can be committed. The checkpoint can be committed when there are no more late messages in transit. To determine when a checkpoint can be committed we use the following observation: when all the messages that were sent in the given checkpoint interval have also been received, there are no more late messages in the system.

An additional variable is added to each process. This variable is used to store the difference between the number of messages sent and the number of messages received by the given process for a given checkpoint. We will refer to this variable as *srDelta* (sent/received delta). Every time when a message is sent, the srDelta is incremented. When a message is received, srDelta is decremented. When the sum of all srDeltas equals to zero, the checkpoint can be committed. When the sum is greater than zero, some messages that are recorded as being sent are not received yet. Note, that srDelta sum is never less than zero.

The resulting algorithm is presented below. The additions to the algorithm from section 4.1 are printed in *italic*.

1. The coordinator starts a new consistent checkpoint by incrementing CN and sending an *Initiate* message that contains CN to each process in the system.

2. Upon receiving an *Initiate* message, a process takes a tentative checkpoint, increments its local CN, and sends a *cpTaken* message to the coordinator, *including its srDelta*.

   If a process receives an application message with a CN greater than its own, it also takes a checkpoint before processing the message.

3. *When the coordinator receives a cpTaken message, it adds the srDelta from the message to its own srDelta.* When the coordinator receives the *cpTaken* message from all processes, *and if its srDelta is zero,* it broadcasts a Commit message.

4. Handling *late messages: when a message arrives whose CN is less than the current CN, a copy of the message is appended to the log file of the checkpoint identified by the message CN. An Update message is sent to the coordinator.*

5. *When the coordinator receives an Update message, it decrements its srDelta. If srDelta equals zero, the coordinator broadcasts a Commit message.*

   Steps 4 and 5 are repeated until all messages that were sent with a given CN are also received.

6. When a pr ocess receives *Commit* message, it makes its tentative checkpoint permanent and discards its previous permanent checkpoint.

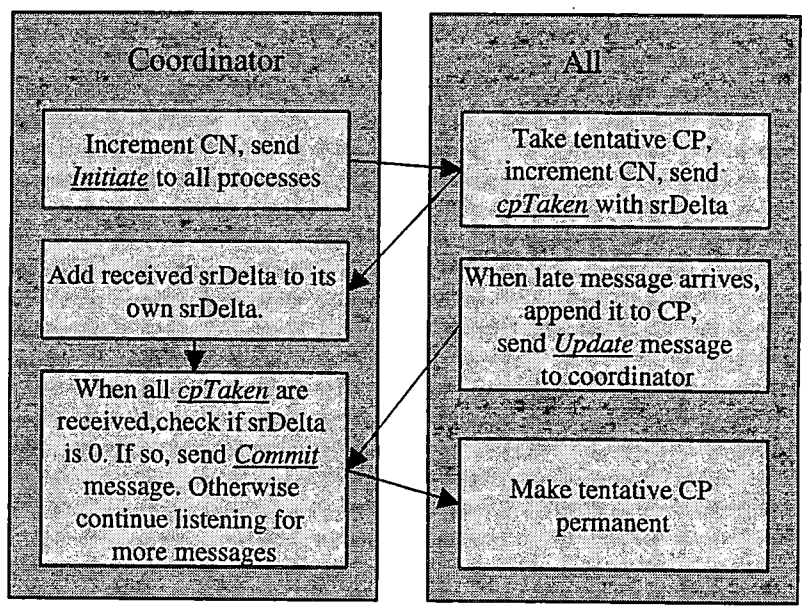


Figure 4

Figure 4 summarizes this algorithm graphically.

4.2.1 Eliminating Inconsistencies

Figure 5 shows how this algorithm captures a consistent system state. Figure 5a shows how received-not-sent inconsistencies are eliminated: message m2 has a higher CN than the process P2. This forces P2 to take a checkpoint before processing m2. Figure 5b shows how sent-not-received inconsistencies are eliminated: message m3 has a lower CN than P2s CN. As a result a copy of m3 (m3a) is appended to checkpoint log identified by m3s CN. During this operation the order in which messages m3 and m2 arrive to P2 is changed. However, these messages are not causally dependent and could arrive in the different order in the regular execution of the program. Therefore global consistency is preserved. This is the only possible change in message order introduced by the algorithm.

We use the following terminology based on tree structures: processes that are being coordinated are called *children*, their coordinator is called the *parent* of the coordinated processes, the processes that have no children are called *leavs*, and the unique process that has no coordinator is called the *root*. The extended algorithm using a hierarchical coordination is shown below, where the added parts are written in *italic*.

1. The *root* process starts a new consistent checkpoint by taking a tentative checkpoint, incrementing CN, and sending an *Initiate* message that contains CN to its *child* processes.

2. Upon receiving an *Initiate* message, a process takes a tentative checkpoint, and increments its local CN. *If the process is a leaf process*, it sends a *cpTaken* message to its *parent*, including its srDelta. *Otherwise, it propagates the checkpoint initiation to its children.*

   If a process receives an application message with CN greater than its own, it also takes a checkpoint before processing the message.

3. *When a process receives a cpTaken message from its child, it adds the srDelta from the message to its own srDelta. When has received the cpTaken message from all its children, it sends the cpTaken message to its coordinator.*

   When the *root* process receives a *cpTaken* message from all its children, and if its srDelta is zero, it broadcasts a *Commit* message to *its children*.

4. Handling late messages: when a message arrives whose CN is less than the current CN, a copy of the message is appended to the log file of the checkpoint identified by the message CN. An *Update* message is sent to the *root process*.

5. When the root process receives an *Update* message, it increments its srDelta. If srDelta equals zero, the root broadcasts a *Commit* message *to its children.*

   Steps 4 and 5 are repeated until all messages that were sent with a given CN are also received.

6. When a process receives a *Commit* message, it makes its tentative checkpoint permanent, discards the previous permanent checkpoint, and propagates the message *to its children*.

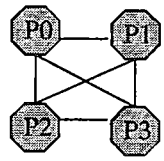
Note that the *Update* message in step 4 is sent directly to the root. If the number of late messages grows proportionally with the number of nodes in the system, then the root process will still be a bottleneck. However, using techniques explained in section 4.2.3 this problem can be avoided.

It is hard to compare the speed of this algorithm with the speed of the centralized version without implementing both algorithms in a large system. However, since both algorithms are non-blocking, the time it takes for the algorithm to complete is not important. Instead we are concerned with the overhead induced by the algorithm. The hierarchical algorithm needs the same number of coordination messages as the centralized one. The only change is that the function of the coordinator is spread over multiple processes. This algorithm also does not perform any additional computations compared with the centralized version. Hence we conclude that the hierarchical algorithm induces as much overhead on the system as the centralized version. The next section presents a mechanism that can be used to organize the processes in the hierarchy.

5 Process Order

In distributed systems it is often desirable to arrange processes in some form of a structure such as a ring, group, or hierarchy. As some processes enter the system and others leave, efficient and transparent schemes to create and maintain these structures are needed. We propose a Process Order mechanism to address this problem.

Each process in the computation has a unique system id. It may consist of two integers: the process id and the IP address. The process id provides uniqueness in the machine, and the IP address provides uniqueness in the network. (If there is only one process per physical node, the IP address is sufficient to provide a unique system id.)



POID	Name	IP
0	P0	20
1	P2	24
2	P1	37
3	P3	44

Figure 6

The processes in the system exchange their system ids during the initialization phase. Then each process sorts all processes by their system ids, which can be represented by integers. After the processes are sorted, each process is given a Process Order Id (POID) according to its place in the sorted list, starting from 0. This information is stored in a *Process Order Table.* An example of a Process Order Table is shown in figure 6.

In order for the Process Order scheme to work, all processes in the system must have the same information about the system topology. All the processes have to have the same set of elements in their PO Table, and in the same order. Since the system is fully connected, it should provide mechanisms to add and remove processes to the system. These mechanisms are sufficient to maintain the correct set of elements in the PO Table. The order of processes is maintained since every PO Table is sorted by unique process ids.

The cost of maintaining the Process Order Table consists of inserting and deleting elements in a sorted list. Even with the most straightforward algorithm this operation is of order $O(n)$. Maintaining the PO Table also requires extra space. In a fully connected system each process has a data structure per each process that contains its name and connection information, such as socket port. The PO Table can be represented as an array of pointers to these data structures. Assuming that unique process identifier is already included in these structures, the pointer in the array is the only additional space required to maintain Process Order.

All the structures, such as groups, rings, trees that are constructed based on the Process Order dont have any additional costs, and can be described by the simple formulas as will be explained next. The following subsections demonstrate how Process Order can be used to address scalability of checkpointing algorithm, for fault detection, and for election of the coordinator.

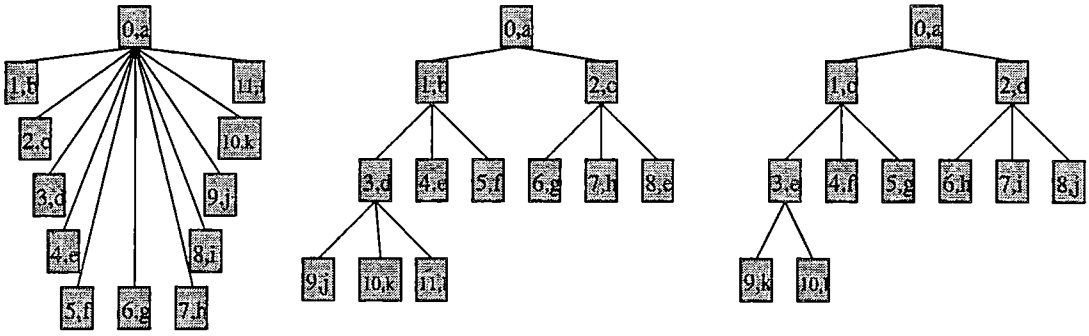
5.1 Process Hierarchy

To address scalability problems in protocols requiring a central coordinator, processes can be arranged in a hierarchy. The Process Order can be used to compute the hierarchy with the formula

$$POID_{coord} = POID_{self} \, DIV \, K,$$

where K is the number of coordinated processes.

Using this scheme each process can identify its coordinator without exchanging any messages with other processes. Figure 7a shows a centralized system with a single coordinator. Figure 7b illustrates how this scheme works when the number of coordinated processes is set to three. The number in each process name is



$POID_{coord} = POID_{self} DIV K$
$K = 3$

$POID_{coord} = POID_{self} DIV K$
$K = 3$

a) centralized system

b) hieraarchical organization

c) process b exited the system

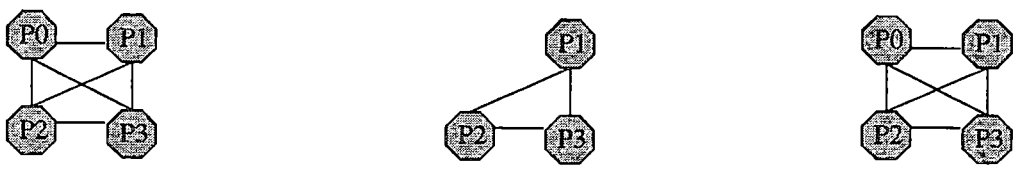
Figure 7

the process POID, and the letter is a symbolic name of the process. This scheme, as any other structure based on Process Order, can dynamically adapt to the changes in the system topology. Figure 7c shows the situation where process b exits the computation. The system automatically rearranges itself in the new hierarchy.

5.2 Fault Detection

For the purposes of fault detection, the participating processes are generally arranged in a logical ring. Each process sends a heartbeat message to the next process in the ring. Depending on the communication protocol, failures can be detected either by the sender of the heartbeat message or by the receiver. TCP/IP generates an interrupt on the sender side if the receiver socket is down. Otherwise failures can be detected by the receiver, based on a timeout. Processes can be arranged into a ring with Process Order using the formula

$$POID_{receiver} = (POID_{sender} + 1) mod\ n,$$



a) system before the failure

POID	Name	IP
0	P0	20
1	P2	24
2	P1	37
3	P3	44

b) P0 exited

POID	Name	IP
0	P2	24
1	P1	37
2	P3	44

c) P0 restarted on another machine

POID	Name	IP
0	P2	24
1	P1	37
2	P0	38
3	P3	44

Figure 8

where n is a number of nodes ($n > 1$)

5.3 Electing a new coordinator

Some protocols used during distributed system execution require a coordinator. If a coordinator leaves the system, a new coordinator needs to be elected. A simple rule for electing a new coordinator using the Process Order is Coordinators POID = 0. When a process receives a notification that another process has left the system, it updates its Process Order Table, removing the exiting process from it. If this was the coordinator, the process next in line moves up to the top position. It is assigned POID = 0 and those becomes the new coordinator. Similarly, a new process can be added to the table and could displace the current coordinator if its system id is smaller than the coordinators.

Figure 8 illustrates this election mechanism. Figure 8a shows the system before any failure. Figure 8b shows how a new coordinator is chosen in the case when a process exits the system. Figure 8c shows how a coordinator is elected when a failed process is restarted on a different machine. This scheme does not require any additional messages to be exchanged.

6 Summary

In this paper we have presented a new checkpointing algorithm for systems that use reliable communication channels. The algorithm is non-blocking, has minimal stable storage requirements, requires $O(n + m)$ communication messages, and minimal message logging. It does not require the channels to deliver messages in FIFO order. This algorithm is scalable, simple, efficient, transparent to the user, and facilitates fast recovery. The combination of these characteristics makes this algorithm attractive for practical implementation.

We also presented a Process Order mechanism that facilitates organizing processes in a fully connected system into different logical structures to solve various problems. These include scalability, failure detection, and coordinator election. The Process Order mechanism groups the processes transparently to the user, and can automatically adjust to changes in system topology. The important characteristic of the Process Order mechanism is that its cost is independent of the number of structures it is supporting.

Using the Process Order mechanism together with the checkpointing algorithm combines the benefits of simplicity and efficiency in coordinated checkpointing. It also eliminates the drawbacks of a central coordinator, i.e., coordinator bottleneck in large systems and a single point of failure.

The work presented here has been partially implemented in the MESSENGERS system [3].

References

1. E. N. Elnozahy, D. B. Johnson, Y. M. Wang. A Survey of Rollback-Recovery Protocols in Message Passing Systems, *Technical Report CMU-CS-96-181*, School of Computer Science, Carnegie Mellon University, October 1996

2. K.M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Systems*, 3(1):63-75, February 1985

3. The MESSENGERS system. *http://www.ics.uci.edu/~bic/messengers/messengers.html*

4. Ten H. Lai and Tao H.Yang. On Distributed Snapshots. *Information Processing Letters*, 25:153-158, May 1987

5. Carol Critchlow and Kim Taylor. The inhibition Spectrum and the Achievement of Causal Consistency. *Technical Report TR 90-1101*, Cornell University, February 1990

6. Kai Li, Jeffrey F. Naughton, James S. Plank. Checkpointing Multicomputer Applications. In *Proc. of the 10th Symposium. on Reliable Distributed Systems,* pages 2-11, 1991

7. R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Trans. Software Eng.*, SE-13(1):23-31, January 1987

8. Yong Deng and E. K. Park. Checkpointing and Rollback-Recovery Algorithms in Distributed Systems. *Journal of Systems and Software*, 25:59-71, 1994

9. E. N. Elnozahy and W. Zwaenepoel. Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast output commit. *IEEE Trans. on Computers Special Issue On Fault Tolerant Computing*, 41(5):526-531, May 1992

10. E. L. Elnozahy, D. B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In *Proc. of the 11th Symposium on Reliable Distributed Systems*, pages 39-47, October 1992

11. Y. Tamir, C.H. Sequin. Error recovery in multicomputers using global checkpoints. *In Proc. of the 13th Int. Conf. On Parallel Processing*, pages 32-41, 1984

12. P.Y. Leu, B. Bhargava. Concurrent Robust Checkpointing and Recovery in Distributed Systems. In *Proc. of 4th IEEE Int. Conf. On Data Eng.*, pages 154-163, 1988

13. Z. Wojcik, B. E. Wojcik. Fault Tolerant Distributed Computing Using Atomic Send Receive Checkpoints. In *Proc. of 2nd IEEE Symp. On Parallel and Distributed Processing*, pages 215-222, 1990

14. F. Cristian, F.Jahanian. A timestamp-based checkpointing protocol for long-lived distributed computations. In *Proc. of 10th Symposium on Reliable Distributed Systems*, pages 12-20, 1991

15. P. Ramanathan, K.G. Shin. Checkpointing and rollback Recovery in a distributed system using common time base. In *Proc. of 7th Symposium on Reliable Distributed Systems*, pages 13-21, 1988

16. L. M. Silva and J. G. Silva. Global checkpointing for distributed programs. *In Proc. of the 11th Symposium on Reliable Distributed Systems*, pages 155-162, 1992

17. J. Leon, A. L. Fisher, and P. Steenkiste. Fail-Safe PVM: A portable package for distributed programming with transparent recovery. *Tech. Rep. CMU-CS-93-124*, Carnegie Mellon Univ., February 1993

18. D. Briatico, A. Ciuffoletti, and L. Simoncini. A distributed domino-effect free recovery algorithm. In *Proc. of the 4th Symposium on Reliable Distributed Systems*, pages 207-215, October 1984

19. Eugene Gendelman, Lubomir F. Bic, and Michael B. Dillencourt. Abstract Checkpoints: an Application-Transparent, Platform-Independent approach. In submission. *http://www.ics.uci.edu/~egendelm/publications.html*