ICS

TECHNICAL REPORT

Process Interconnection Structures in Dynamically Changing Topologies

Eugene Gendelman
Lubomir F. Bic
Michael B. Dillencourt

UCI-ICS Technical Report No. 00-27
Department of Information and Computer Science
University of California, Irvine, CA 92697-3425

August 17, 2000

Information and Computer Science
University of California, Irvine

Process Interconnection Structures in Dynamically Changing Topologies

Eugene Gendelman Lubomir F. Bic Michael B. Dillencourt
Department of Information and Computer Science
University of California
Email:{egendelm, bic, dillenco}@ics.uci.edu

Abstract

Centralized coordination protocols are simpler and more efficient than distributed ones. However, as a distributed system gets large, the bottleneck of the central coordinator renders protocols relying on centralized coordination inefficient. To solve this problem, hierarchical coordination can be used, where performance degrades logarithmically with the number of participating processes.

In this paper we present a mechanism that automatically organizes processes in a hierarchy and maintains the hierarchy in the presence of node failures, and incremental addition and removal of processes in the system. The new topology resulting from a change is computed by each process locally, without having to broadcast the entire topology to all processes. The proposed scheme can concurrently support multiple logical structures, such as a ring, a hypercube, a mesh, or a tree. It supports total order of broadcasts and does not rely on any specific system features or special hardware.

1. Introduction

Distributed systems consisting of a network of workstations or personal computers are an attractive way to speed up large computations. There are several coordination protocols that must be used during the system execution. Among these protocols are those responsible for checkpointing [1], stability detection [3], and maintaining a global virtual time [4]. Usually, protocols that involve a central coordinator are more efficient, as they require fewer communication messages than distributed protocols, and are simpler to construct. At some point of the centralized protocol a coordinator broadcasts information to other processes and receives an acknowledgement message. As a system gets large, a bottleneck of the central coordinator significantly slows down the execution of the centralized protocols, sometimes even making them unusable. A hierarchical coordination can be used instead of a centralized one [3,5]. Hierarchical protocols have the efficiency and simplicity of the centralized protocols and scalability of the distributed protocols. In this paper we present a mechanism, called Process Order, for building and maintaining a hierarchy of processes in a dynamically changing distributed system.

The rest of this paper is organized as follows. As a motivation for this work, section 2 presents performance evaluation of centralized and hierarchical broadcasts. Section 3 describes the Process Order mechanism. Section 4 shows how Process Order operates in a dynamically changing failure-free system. Section 5 describes Process Order in the presence of failures. Section 6 presents

possible extensions and optimizations to the to the basic scheme. And section 7 concludes this paper.

2. Performance of Centralized and Distributed Coordination

This study measures the performance of a broadcasting protocol, which is used in many coordination protocols [2-5,7]. It consists of broadcasting the message, and receiving all the acknowledgement messages back.

Experiments were conducted on a cluster of Sun workstations running Solaris 2.5.1 and Solaris 2.7. The workstations were located in two neighboring subnets. Within each domain the workstations were connected with a 10 Mbps shared Ethernet with collision domain less or equal to twelve. Domains were connected by 100 Mbps fiber with Cisco router 5500. The experiments were run at night, so the fluctuations in the workstations' load were minimal.

Inter-process communication was implemented with TCP/IP, as it provides guaranteed message delivery, which is essential for many coordination protocols. The application program was implemented in Java.

Each process in the system consists of two threads: an application thread, and a coordination thread. An **application thread** computes a synthetic grid application. Processors form a 2-dimentional logical grid, where every element of the grid is connected with its four nearest neighbors.

Each grid element executes the same function, which takes in an array of doubles and performs a simple arithmetic modification to each array element. Then a message with a user-defined size is sent to the four neighbors, and a similar message is received from the four neighbors. This sequence is repeated for the duration of the experiment. Socket connections in this thread are established at the beginning of the execution, and closed only when the program terminates.

The **coordination thread** running in parallel to the application thread is executing broadcasts. The centralized and the hierarchical broadcasts were implemented. In the **centralized broadcast** all the processes in the system are divided into a coordinator process and coordinated processes. The coordinator process spawns two threads: one for sending and one for receiving. A sending thread broadcasts a message to all the coordinated processes, and a receiving thread receives all the responses. A new cycle of broadcasting can not be started until all the responses to the current broadcast are received. The coordinated processes have only one thread; it receives a message from the coordinator, and replies back.

In the **hierarchical broadcast,** *every* process spawns two threads: one for sending and one for receiving. The processes form a logical binary tree, and the broadcast travels down the tree and back. A node sends a reply to its parent only after it receives replies from all its children. A new cycle of broadcasting can not be started until the previous is completed.

There is a limit on the number of socket connections that could be open at the same time. Solaris 5.1 allows only 64 file descriptors open at any time. Considering this, a new connection is set up for every send and receive for the broadcasting thread of the coordinator in the centralized broadcast. This is not necessary in the case of the hierarchical broadcast. However, hierarchical broadcasting was measured both with and without reconnections. These results are used later in this paper.

A time of one broadcast for each protocol and for each system configuration was measured. The measured time includes overhead for starting the Java process and initializing global variables. In the presented results the time of the first iteration is subtracted from the total execution time. The measurements for each experiment were averaged over the set of several runs.

Figure 1 presents running 100 broadcasts for each protocol on the sets of 15, 16, and 63 processing elements (PEs). As the number of PEs increases, the advantage of hierarchical broadcast becomes obvious. With 63 PEs the hierarchical broadcast with reconnection (H1) takes 19 times less time than centralized broadcast (C), and hierarchical broadcast without reconnection (H2) is 5 times faster than H1.

	63 PE	16 PE	15 PE
C	463.2	39.6	39.7
H1	24.2	14.1	7.5
H2	4.8	3.5	1.3

100 broadcasts (sec)
Figure 1

For the centralized broadcast in the first two columns of figure 1 the number of processors decreased by a factor of four, and the running time decreased by a factor of almost twelve. This reflects the difference in load on the central coordinator induced by 63 and 16 PEs.

There was no noticeable difference in time taken by the centralized broadcast when the number of PEs decreased from 16 to 15. For hierarchical broadcast the difference in performance between columns one and two is similar to that of columns two and three. The reason for this is that in both cases, i.e., when moving from 63 PEs to 16 PEs and when moving from 16 PEs to 15 PEs, the depth of the tree is decreased only by one level. As all the nodes at the same depth of the tree broadcast to their children in parallel, the broadcasting time grows with the depth of the tree, and not with the number of PEs.

Modifications to the application thread including changing the ratio of computation to communication, and changing the size of the exchange message had no effect on performance of the broadcasting protocols.

In summary, our experiments confirm the intuition that centralized broadcasting is slow for large systems and that hierarchical broadcasting scales well as the system size increases.

3. Process Order

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.)

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 for a system with 4 PEs is shown in figure 2.

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

Process Order Table
Figure 2

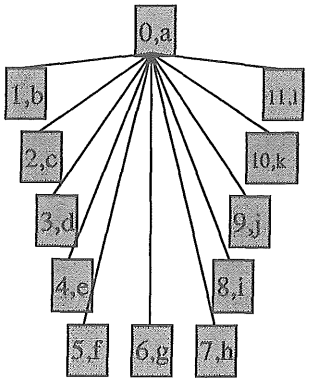
The processes can be arranged in a hierarchy implicitly with the Process Order using the formula

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

where K is the maximum number of processes coordinated by any single process.

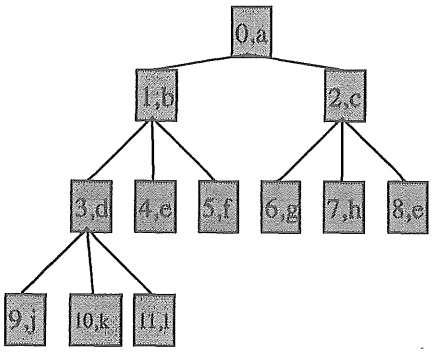
Using this scheme each process can identify its coordinator without exchanging any messages with other processes. Figure 3a shows a centralized system with a single coordinator. This can be modeled as a 2-level hierarchy with K equals to the total number of processes. In this and all following examples processes are identified by a letter, and then assigned POID in alphabetical order. Figure 3b illustrates how this scheme works when the number of coordinated processes is set to three. The number in each process name is the process POID, and the letter is a symbolic name of the process. This hierarchy can dynamically adapt to the changes in the system topology. Figure 3c shows the situation where process "b" exits the computation. The system automatically rearranges itself in a new hierarchy.

The use of this hierarchy can easily be extended to a more general broadcasting mechanism, where all the processes in the system can broadcast messages. To do this, the initiator of the broadcast sends its message to the root coordinator, which then propagates the message down the hierarchy. As TCP/IP provides reliable FIFO channels, this broadcasting mechanism supports a total order of broadcasts.



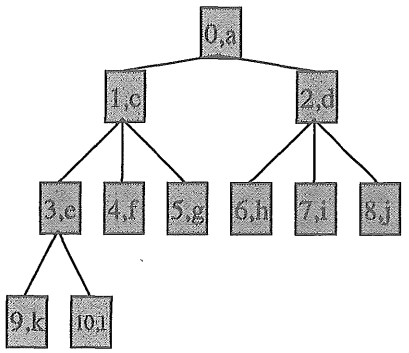
$POID_{coord} = POID \quad DIV\ K$
$K = 12$

a) centralized system



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

b) hierarchical organization



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

c) process "b" exited the system

Figure 3

4. Process Order in a dynamically changing failure-free system

This section describes how the consistency of the Process Order mechanism is preserved when processes can join or leave the system at run time. We assume that all deletions and insertions of processes are initiated by special modification messages sent to the current coordinator process.

Definition 1: The system modification is successful, iff

1. ***All broadcasts initiated before a modification message is delivered to the old PE set.***
2. ***All live processes receive the modification message.*** This includes any process being deleted or inserted
3. ***All broadcasts initiated after the modification message are delivered to the new PE set.***
4. ***All broadcasts are delivered in total order.***

When a process is joining the system it sends a *SystemJoin* message to the root coordinator containing its ip address. This is illustrated in figure 4a, as node B joins the system. The root updates its Process Order Table (POTable), and sends a *POModify* message to its newly calculated children, as shown in figure 4b, where *POModify* messages shown by thick arrows. *POModify* message includes the information about the new process. After receiving *POModify* message a process modifies its POTable. If the children list changed, the process establishes connections with its new children and forwards the *POModify* to them (fig 4c).

Similarly, when a process is exiting the system, it sends a *SystemLeave* message to the coordinator, as does node X in figure 5a. The coordinator modifies its POTable and broadcasts *POModify* message to its new children (fig 5b). This is repeated down the tree, as shown in figure 5c. The old parent of the leaving process sends *POModify* message to its new children, as well as to the leaving process. In figure 5c B signals X to leave.

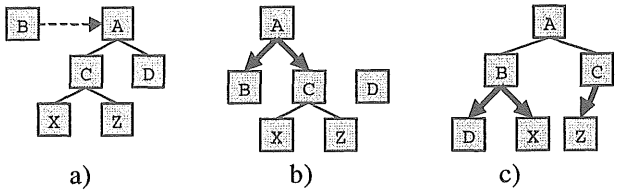


Figure 4. Adding node B

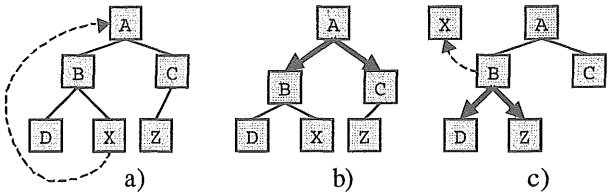


Figure 5. Removing node X

Each process keeps a modification log (ModLog), which is a collection of the received *POModify* messages. Each process also keeps the log of the broadcasts (BLog). The modification algorithms will be clarified, as their correctness is shown.

Proof of Correctness

1. ***All broadcasts initiated before a modification message are delivered to the old PE set.***

As the topology of the system changes, messages that would normally arrive in the FIFO order, can be interchanged, as shown in Figure 6. In this example (fig.6a) broadcast **b1** is started before adding node B, which is initiated by a message **m** (fig 6b). This is followed by broadcast **b2** (fig 6c). Node D receives **b1** from A, and **m** and **b2** from B. Theoretically **b1** could reach D after **m** and even after **b2**. In this case D would broadcast **b1** only to Y, and b1 would never be delivered to node X (fig 6d).

This problem is fixed by appending the ModLog of the root coordinator to the broadcasted message. This allows processes to reconstruct the hierarchy as it was at the time of the broadcast. As ModLog of A did not contain

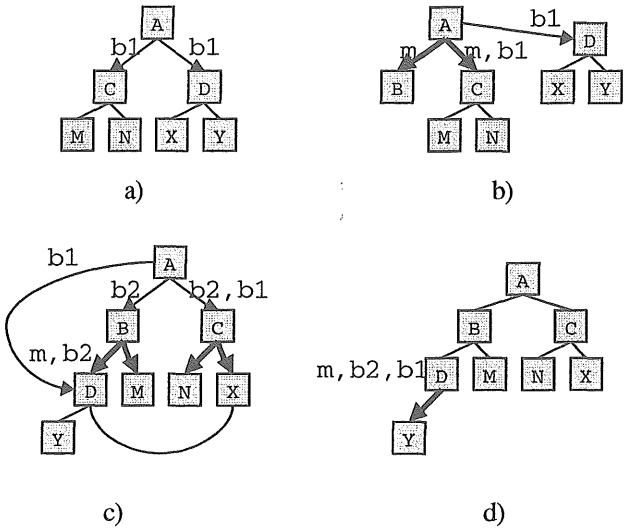


Figure 6. Broadcasts while adding node B.
b1, b2 - broadcasts, m - modification message

the addition of node B, D will also send b1 to X.

2. All live processes receive the modification message.
Each process processes the *POModify* message, modifies its POTable, and then broadcasts *POModify* message to its new children. As this is done consistently by every process, every process in the new hierarchy receives the modification message. If a process is being deleted then it receives the *POModify* message from its original parent.

3. All broadcasts initiated after the modification message are delivered.
When a new broadcast is initiated, it traverses a new hierarchy built by *POModify* messages. As TCP/IP provides FIFO channels, a broadcasting message cannot reach its destination before the *POModify* message, and therefore the broadcast is delivered to the whole new hierarchy.

4. All broadcasts are delivered in total order.
In a modification-free execution all broadcasts are delivered in total order. Therefore, all broadcasts that happened before a modification, and all broadcasts sent after a modification are delivered in the correct order. However, broadcasts sent before the modification (point 1 above) could mix with broadcasts initiated after the modification, as was shown on figure 6.

To solve this problem, the root coordinator assigns all the broadcasts a unique, monotonically increasing broadcast index (BI). Each process knows the BI of the last message it received, and therefore can postpone processing out-of-order messages. Reliable communication channels guarantee the delivery of all messages, or failure will be detected otherwise. Therefore, deadlock is not possible.

5. Process Order in the presence of failures

5.1 Failure detection

TCP/IP generates an interrupt on the sender side if the receiver socket is not responding. This feature is used to detect process failures. The participating processes are arranged in a logical ring using Process Order. Each process sends a heartbeat message to the next process in the ring. Processes are arranged into a ring with Process Order using the formula

$$POID_{receiver} = (POID_{sender} + 1) \bmod n, \textit{ where } n \textit{ is a number of nodes } (n > 1)$$

5.2 Failure recovery

Consistency of the Process Order mechanism in the presence of failures depends on the recovery algorithm used in the system. If the rollback-recovery algorithm presented in [6] is used, the consistency of the Process Order is preserved automatically. In this scheme, once failure is detected, the whole system rolls back and recovers from the previously saved consistent system state. When the recovery algorithm is finished, all the processes aware of all other live processes in the system, and therefore Process Order tables, are consistent. This recovery scheme might be good for applications requiring barrier synchronization, individual-based simulations relying on the virtual time, and grid applications.

For other applications this recovery scheme might produce large overhead, especially in large systems. In such cases recovery protocols that restart only the failed process are used. One way to provide such a recovery is by using a causal logging protocol [8, 12]. The rest of this section describes how Process Order consistency is preserved in case of such a recovery.

5.2.1 Non-root node failures

The node failures could be divided into three types: 1. Failures occurring before processing a broadcast message, which was received, but not propagated down the hierarchy. 2. Failures occurring in the middle of the broadcast, when the process sends a message to some of its children, and then fails. 3. Failures occurring after the broadcast, when node is not participating in the broadcast. The *POModify* messages are treated in the same way as other broadcast messages.

The idea is to merge the first two types of failures with the simpler third type. This is done by comparing the BI index of node's parent to the BIs of the node's children. If the failed node's child misses some messages received by the failed node's parent, these messages are sent to that node. After this is done, the failure could be considered belonging to type 3. Multiple failures are handled in a similar way. In figure 6d if node B fails, the BIs of D and M are compared to BIs of A. If D also fails, the BIs of Y and M are compared to A.

Type 3 failures are handled in the following way. After failure is detected, the coordinator is notified, and it acts in the same way as if the failed process willingly leaves the system: the coordinator broadcasts *POModify* message to its children.

If a process during a broadcast discovers that its child in the hierarchy is down, it sends a notification message to the root coordinator, and propagates the broadcast to the children of the failed process. For example, if node D in figure 6d failed and B tried to send it a broadcast, B would notify A of D's failure and propagate the broadcast to Y and to M. As a result all the processes, except the failed one, receive the broadcasted message. The same rule works in the presence of multiple failures.

5.2.2 Coordinator failures

The failures of the coordinator could be divided into the same three types as the failure of any other node. Type 3 failure is the same in both cases but the recovery from type 1 and 2 failures is different, since the coordinator has no parent.

Type 1 failure occurs when some node sends a message to the coordinator in order to initiate its broadcast. If the coordinator fails before broadcasting the message, then the message will be lost. To prevent this, the initiator of the message is responsible for making sure the message is broadcasted. The initiator appends a unique monotonically increasing broadcast number to the message. When the initiator is notified of the coordinator failure, it checks if it received all the messages that it initiated. If not, then it sends a message to the new coordinator asking to repeat the broadcast.

Type 2 failures occur when the coordinator fails during the broadcast. To circumvent this, a new coordinator compares its BIs with the BIs of all the children of the failed coordinator. If some of the children miss a certain message, it is rebroadcasted to that particular part of the old tree. Each of the rebroadcasted messages contains the history of changes to the topology that happened by the time the original broadcast was made. This way each node in the system can figure out its descendants in the tree before the failure, and the whole old hierarchy is reconstructed.

The type 3 failures of the coordinator are handled in the same way as the type 3 failures of other nodes, except that the failure notification message is sent to the new coordinator. The new coordinator is always the one with the lowest POID.

5.2.3 Node recovery

To preserve the total order of broadcasts, a restarted process must also receive all the broadcast messages it missed while being down. Comparing the BIs of the restarted process with BIs of the

coordinator determines the messages missed by the failed process. When the failed process restarts, these missed broadcast messages are supplied to the failed process along with other logged messages needed for the recovery. When the failed process is ready to rejoin the system, the total order of broadcasts in the system is preserved.

The ModLog and BLog are periodically cleared along with traditional causal logging protocol logs during the capture of a consistent system snapshot, or by a specialized message.

6. Extensions to the basic scheme

6.1 Broadcasting without total order

If the total order of broadcasts is not a necessity, the initial message does not need to go through the coordinator. Instead, the sender process becomes a root of another broadcast tree. The POID of the coordinator is sufficient for all other processes to construct the desired tree on the fly. Figure 7 shows how this tree is constructed with node D as a coordinator. The indices 0-5 are simply rotated so that D is assigned the POID zero. This results in the tree where D is the root. Both trees, with coordinators A and D, respectively can coexist. All topology modifications are still going through the tree with the coordinator A. Modifications to the topology do not cause inconsistencies in D's tree as the sender's ModLog is attached to the message, so that every node can calculate its children, as was shown in section 4. This scheme reduces message traffic and, most importantly, it removes the bottleneck resulting from a central coordinator.

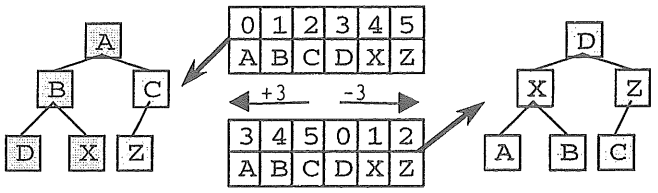


Figure 7. Constructing tree with D as a coordinator

6.2 Possible optimizations

Modifying the system topology also requires restructuring of the tree, which might involve breaking old and establishing new TCP/IP connections. The penalty of restructuring the tree is discussed in section 2. As was shown in section 4, appending the list of changes to the broadcast

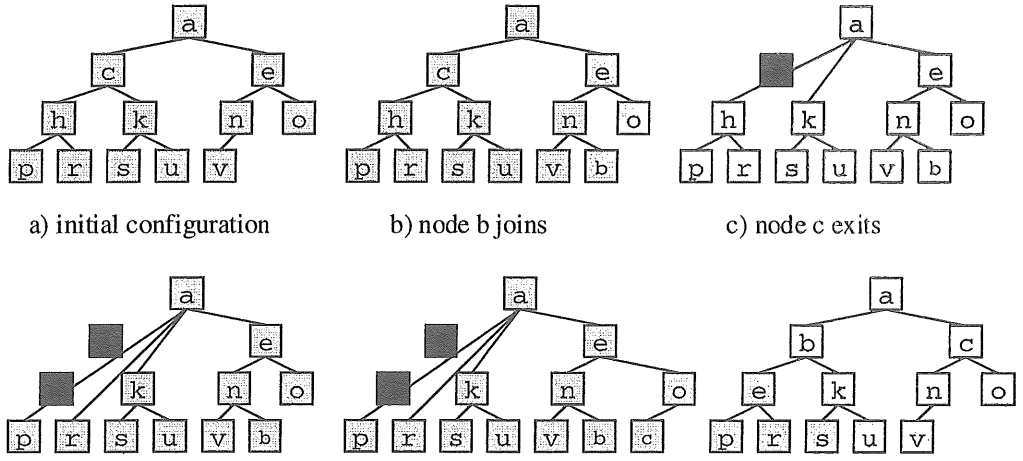


Figure 8. Lazy update

allows delivery of the message to intended recipients. This fact can be used to optimize restructuring. Instead of updating the tree with every change, the tree is updated only occasionally (lazy update). Figure 8 illustrates this approach. When a node joins the system, it is added at the leaf level of the tree (fig 8b and fig 8e). When a node exits the system, the parent of the exited node takes responsibility for broadcasting the message to the exited node's children (fig 8c and fig 8d). These modifications are recorded in a special log. When this log is cleared, together with ModLog, the tree is recomputed, taking into account all the changes in the logs (fig 8f).

For the scheme presented in section 6.1, where the broadcasting tree changes for each broadcasting process, the cost of reconnecting the processes could be high. A possible solution is to swap the broadcasting node with the original root node, as shown on figure 9.

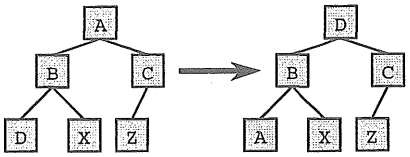


Figure 9. Constructing tree with D as a coordinator

Another optimization could be implemented when processes are distributed through the WAN. The hierarchy could be built in two steps to minimize the communications between different parts of the network. First a hierarchy is built between hosts in different subnets, and then within the subnets. This is possible, as the location of the machine is specified in its IP address.

7. Conclusion

The Process Order mechanism is a flexible and a simple tool for creation and maintenance of logical structures in distributed system. Its main advantage over other approaches is that nodes only need to have a list of currently active processes. Each node can then determine the complete topology of the system locally, using a formula. Thus the topology of the system never needs to be broadcasted to other nodes. The approach can be implemented using the TCP/IP protocol, which allows its use on practically any machine. This is an important factor for the systems using non-dedicated workstations.

Aside from the hierarchy, it is often desirable to arrange processes in other logical structures, such as a ring for fault detection, a hypercube [9], to implement stability detection algorithms, a mesh [10], for grid computations, or a group [11], for load balancing. Process Order can be used to construct and maintain these structures in a similar way as the tree. The usability of the Process Order mechanism depends on how well the desired logical structure can be expressed with a formula.

The cost of maintaining the Process Order is a total of n messages when adding or removing a new process to the system, where n is the number of processes. 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 the order of $O(n)$. Maintaining the PO Table also requires minimal extra space: to maintain the unique ids of the participating processes. The cost of maintaining Process Order is independent of how many structures are being concurrently supported by the mechanism.

References

[1] E. N. Elnozahy, D. B. Johnson, Y. M. Wang. "A Survey of Rollback-Recovery Protocols in Message Passing Systems.", *T.R. CMU-CS-96-181*, School of Computer Science, Carnegie Mellon University, Oct. 1996

[2] 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

[3] K. Guo. "Scalable Message Stability Detection Protocols." PhD thesis, Department of Computer Science, Cornell University, 1998

[4] M. Fukuda. "MESSENGERS: A Distributed Computing System Based on Autonomous Objects." PhD thesis, Department of Information and Computer Science, University of California, Irvine, 1997

[5] E. Gendelman, L. F. Bic, M. Dillencourt. "An Efficient Checkpointing Algorithm for Distributed Systems Implementing Reliable Communication Channels." *18th Symposium on Reliable Distributed Systems*, Lausanne, Switzerland 1999

[6] E. Gendelman, L. F. Bic, M. Dillencourt. "An Application-Transparent, Platform-Independent Approach to Rollback-Recovery for Mobile Agent Systems" *20th IEEE International Conference on Distributed Computing Systems*. Taipei, Taiwan 2000

[7] 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

[8] K. Kim, J. G. Shon, S. Y. Jung, C. S. Hwang. Causal Message Logging Protocol Considering In-Transit Messages. In *Proc. of the ICDCS 2000 workshop on Distributed Real-Time Systems*. Taipei, Taiwan 2000

[9] R. Friedman, S. Manor, and K. Guo. "Scalable Stability Detection Using Logical Hypercube." *18th Symposium on Reliable Distributed Systems, Lausanne*, Switzerland 1999

[10] K. Solchenbach and U. Trottenberg. "SUPRENUM: System essentials and grid applications." *Parallel Computing* 7 (1988) pp. 265-281

[11] A. Corradi, L. Leonardi, F. Zambonelli. "Diffusive Load-Balancing Policies for Dynamic Applications". *Concurrency*. January-March 1999.

[12] L. Alvisi, B. Hoppe and K. Marzullo, "Nonblocking and Orphan-Free Message Logging Protocols," *Proceedings of the 23rd Fault-Tolerant Computing Symposium*, pp.145-154, June 1993.