UC Irvine
ICS Technical Reports

Title
Enabling logical structure support in the dynamic distributed environment

Permalink
https://escholarship.org/uc/item/7tw1k6qz

Authors
Gendelman, Eugene
Bic, Lubomir F.
Dillencourt, Michael B.

Publication Date
1999-11-04

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Z
699
C3
no.99-51

ICS

TECHNICAL REPORT

Enabling Logical Structure Support in the Dynamic Distributed Environment

Eugene Gendelman
Lubomir F. Bic
Michael B. Dillencourt

UCI-ICS Technical Report No. 99-51
Department of Information and Computer Science
University of California, Irvine, CA 92697-3425

November 4, 1999

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

Information and Computer Science
University of California, Irvine

SLBAR
Z
699
C3
no. 99-51

Enabling Logical Structure Support in the Dynamic Distributed Environment

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

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

Coordination protocols are an essential part of every distributed system. In general, centralized 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. This solves the scalability problem of the algorithms relying on centralized coordination, since the performance of hierarchical coordination 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 addition and removal of processes in the system. The proposed scheme is simple and general, and can concurrently support multiple logical structures, such as a ring, a hypercube, or a mesh.

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.

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 less communication messages, and are simpler to construct. However, 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. The reason for this is that during certain steps of these protocols the coordinator must broadcast information to all other processes, and receives a reply messages from them.

To deal with this problem 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, at the same time, have the scalability of the distributed protocols.

Even though it is generally agreed that using a hierarchy is a good approach to solving the scalability problem, it is not clear how to build such a hierarchy, and maintain it in a dynamically changing distributed system. Addressing this problem is the main contribution of this paper.

To evaluate the extent to which the use of a hierarchy improves the performance of centralized algorithms in large systems we compared the performance of centralized and hierarchical broadcasts. In [3] K. Guo compares the performance of centralized and hierarchical algorithms detecting a stable property. The experiments were conducted using the NS network simulator [6]. We conducted our experiment in a real distributed environment. We measured the performance of simple broadcasting mechanism, that consist of broadcasting the message, and receiving all the acknowledgement messages. We decided to measure the performance of this broadcasting mechanism, as it is used in many coordination protocols [2-5,7]. The result of the experiments are presented in section 2. Section 3 describes a Process Order mechanism, using which the hierarchy of the processes in the system can be automatically built. Section 4 describes an algorithm used to maintain Process Order consistency in the presence of addition of new nodes to the system as well as node failures. Section 5 discusses APIs to the Process Order library and section 6 concludes this paper.

2 Performance of Centralized and Distributed Coordination

We conducted an experiment to compare the performance of the centralized and the distributed coordination protocols running in parallel with an application in a distributed environment.

2.1 The Environment

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

To implement inter-process communication in our experimental program we considered TCP/IP and UDP, as these protocols are most widely available. TCP/IP was chosen. Even though it has more overhead on creating connection than UDP, it provides guaranteed message delivery, which is essential for checkpointing protocol.

The program was implemented in Java. We also considered implementing it in C, but as connections are rapidly created and destroyed, C programs under UNIX have a large delay period for closing previous connections. Another benefit of using Java is its portability, which is an important factor for utilization of all available resources.

2.2 Application

Our distributed system consists of two threads; an application thread, and a coordination thread. An application thread computes a synthetic grid application. Processors are arranged in a 2-dimentional

```
While(1 == 1){
    compute array[array size];
    send messages (pckt size) to
        neighbors
    receive messages from neighbors
}
```

Figure 1

logical grid. Every element of the grid is connected with its four nearest neighboring elements. Each grid element executes the

same function. Its psudocode is shown in figure 1. Compute() takes in an array of doubles and performs a simple arithmetic modifications to each array element. Then a message with the user-defined size is sent to the four neighbors, and a similar message is received from the four neighbors. Sending and receiving is done through TCP/IP sockets. Socket connections in this thread are established at the beginning of the execution, and closed only when the program terminates.

Sending Thread:
```
for (number of receivers){
   Open Connection
   Send message
   Close Connection
}
```
Receiving Thread:
```
for (number of receivers){
   Accept Connection
   Receive message
   Close connection
}
```

a) Coordinator Process

```
{
   Accept Connection
   Receive message
   Close connection

   Open Connection
   Send message
   Close Connection
}
```

b) Coordinated Processes

Figure 2. Centralized broadcast

2.3 Centralized Broadcast

The coordination thread running in parallel to the application thread is executing periodic broadcasts. We implemented three broadcasting strategies. A centralized strategy, and two hierarchical strategies. The implementation of the centralized broadcasting is summarized in figure 2.

All the processes in the system are divided on coordinator process and coordinated processes. The broadcasting thread of 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 broadcasting thread of all coordinated processes have only one thread, where they receive a message from the coordinator, and reply back.

Since there is only a small number of socket connections that could be open at the same time, a new connection is set up for every send and receive. On the Solaris 5.1 only 64 file descriptors can be open at any time. Considering that our application thread consumes several socket connections, and that there are other applications running on this machine, we want to keep the number of the open file descriptors used by the broadcasting to the minimum.

2.4 Hierarchical Broadcast

In the hierarchical broadcast the broadcasting thread of every process spawns two threads: one for sending and one for receiving. Again, as in the centralized broadcasting, we want to minimize the number of open socket connections, and so every time a message is sent or received, a new connection has to be established. A new cycle of broadcasting can not be started until the previous is completed.

Processes participating in the hierarchy can be divided into three groups: a node that has no coordinator, or a root node, nodes that have both coordinator and children, or middle nodes, and nodes that have a coordinator and no children, or leaf nodes. The pseudocode for each of these groups is shown in figure 3.

```
Root node:   open, send, close to all children
        accept receive close from all children

Middle node: accept, receive, close from parent
        open, send, close to all children
        accept receive close from all children
        open, send, close to parent

Leaf node:  accept, receive, close from parent
        open, send, close to parent
```

Figure 3. Hierarchical broadcast

A node sends a reply to its parent only after it receives replies from all its children.

2.5 Experiment results

In the first set of experiments we compared the performance of centralized broadcasting with a hierarchy, where each node has a maximum of two children. In such a hierarchy the maximum number of open connections per process is three: one to connect to the parent in the hierarchy, and two to connect to the children. These two extra socket connections could be acceptable for many applications. For this reason, in addition to the implementation described in section 2.4 we implemented a broadcasting scheme, where socket connections are established only once, and then used for the rest of the run. We call this implementation SuperH. As we have only a limited number of workstations for our experiment, we run our broadcasting routine multiple times.

First we measured the time of one broadcast for each protocol and for each system configuration. The measured time includes overhead for starting the Java process and initializing global variables. In the following experiments we subtract the time of the first iteration from the total execution time of the program to get the time spent on broadcasting only.

In all experiments we used the same workstations for each of the three implemented protocols. The measurements for each experiment were averaged over the set of several runs.

Figure 4 presents the times for running 10 iterations of the broadcast on 63 processing elements (PE). The hierarchical broadcasting

	63 PE
Centralized	44.6
Hierarchical	2.1
SuperH	0.6

10 iterations on 63 processing elements

Figure 4

takes 21 times less time than centralized broadcasting. SuperH is 3.5 times faster than the Hierarchical implementation. It is worth noting that a large part of the speedup came from arranging processes in the hierarchy, and only a small part is due to the fact that in SuperH the overhead of establishing socket connection is eliminated.

Even though all broadcasting strategies were run for ten iterations, the simulated number of processes participating in the broadcast is different. In centralized broadcast we simulated a system with 630 PEs. However,

	63 PE	16 PE	15 PE
Centralized	07:43.2	39.6	39.7
Hierarchical	24.2	14.1	7.5
SuperH	4.8	3.5	1.3

100 iterations. 63,16,and 15 processing elements

Figure 5

the bottleneck of the central coordinator induced by the 63 PEs is much smaller than one induced by 630 PEs, as will be demonstrated later. Theoretically, with hierarchical broadcasts we simulate a system with 2^{51}-1 PEs. 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. Figure 5 illustrates these arguments.

For the centralized broadcast in the first two columns of figure 5 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.

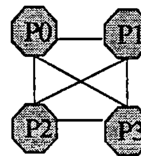
We also ran experiments modifying the behavior of the application thread, described in section 2.2. We modified the ratio of computation to communication, but these modifications had no effect on performance of the broadcasting protocols.

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

3 Process Order

3.1 Introduction

In this section we describe how to build and maintain the participating processes in the hierarchy. This becomes complicated, as processes continuously enter and leave the system. We propose a Process Order mechanism to deal with this problem.



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

Figure 6

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.)

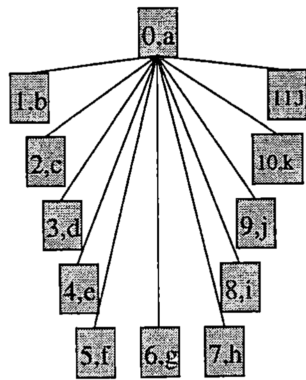
The processes in the system exchange their system ids during the initialization phase. Then each process sorts all processes by their system ids, which can be represented by integers. After the processes are sorted, each process is given a Process Order Id (POID) according to its place in the sorted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

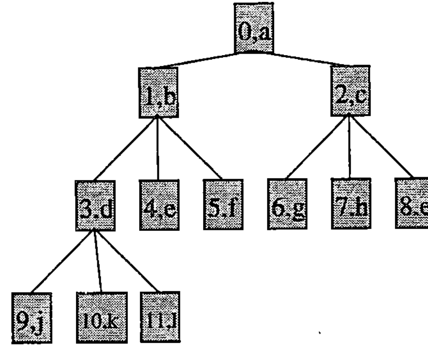
3.2 Building a Hierarchy

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

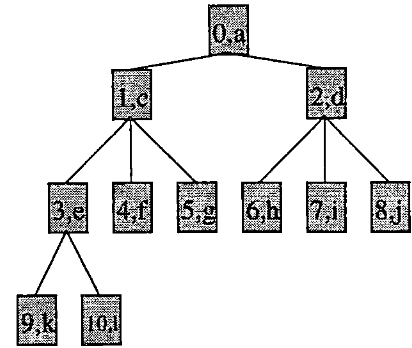
$$POID_{coord} = POID_{self}\ DIV\ K,$$

where K is the 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 7a shows a centralized system with a single





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



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

a) centralized system

b) hieraarchical organization

c) process "b" exited the system

Figure 7

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 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 7c shows the situation where process "b" exits the computation. The system automatically rearranges itself in a new hierarchy. The same happens when coordinator (root) exits, which solves a single point of failure problem.

3.3 Hierarchy on the WAN

When processes are distributed through the WAN, a 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.

4 Maintaining Process Order Consistency

In order for the Process Order scheme to work, all processes in the system must have the same set of elements in their PO Table. In this section we discuss the necessary operations to keep PO Tables consistent as processes enter and exit the system. First we present an algorithm that preserves PO Table consistency in the failure-free environment.

4.1 Basic Algorithm

When a process is joining or exiting the system

1. Message is sent to the root coordinator containing IP and PID of the process that is being added/removed. Root updates its PO Table, and, if the process is being added, it sends a copy of its PO Table to the new process. Then root sends the *POModify* message to its children. This message includes the information about the process, as well as whether the process is being added to or removed from the system.

Until the update is finished, the Process Order is inconsistent, so root does not broadcast any application messages while updating Process Order is in progress.

2. After receiving *POModify* message a process modifies its PO Table. If children list changed, (as it did for the process *a* in figure 7), the process breaks connections with its old children and establishes connections with its new children. Then the process broadcasts *POModify* message to its children.

   When a leaf process receives *POModify* message, it replies with *POModifyAck* message to its parent.

3. When a node receives *POModifyAck* from all its children, it sends *POModifyAck* message to its parent.

   When root receives *POModifyAck* message from all its children, the use of the hierarchy is resumed.

4.2 Fault Tolerant Algorithm

If failures are possible, then applying the algorithm from section 4.1 can produce inconsistencies. For example, assume a process is joining the system. Root broadcasts a *POModify* message, and fails in the middle of the broadcast. The message is propagated along one of the branches, but not along the other. If a new root is selected, using the formula presented in section 3.2, the system would continue executing with different processes having different PO Tables.

Another, more common scenario, is the following: when a new process is being added to the system one of the nodes fails. As a result, coordinator never receives the *POModifyAck* from all its children, which may result in a deadlock or PO Table inconsistency, depending on how failures are handled.

To deal with these problems, each process keeps an additional data structure *POPendingList* to maintain a list of the processes that are currently being added to or removed from the PO Table. Each entry in the POPendingList contains process information and whether it is being added to or removed from the system. When broadcasting *POModify* message, a process attaches its POPendingList to it. When a process receives such a message, it compares the received POPendingList with its local POPendingList. If the local POPendingList is a subset of the received POPendingList, then changes are made to the PO Table, and the message is propagated to the process's children. Otherwise, the process computes the union of the two POPendingList structures and sends the *POModifyNew* message to the root. When root receives *POModifyNew* message, it compares the attached POPendingList with its local POPendingList. If the lists are not equal, root computes the union of the two POPendingList structures and sends a new *POModify* message to its children.

When multiple processes fail, it is possible, that a *POModifyNew* message will be sent to the non-root process. There are two possible scenarios: first, a receiver process knows that it is not a root process, and second, it does not know. If the process knows that it is not a root process, it simply forwards the message to what it thinks is the root process. If the process thinks it is a root process, it proceeds as a root. Then the inconsistency will be detected somewhere down the hierarchy, and *POModifyNew* message will be sent.

Each *POModifyAck* message also includes the POPendingList of its sender. When the process receives *POModifyAck* message, it

compares the received POPendingList with its local POPendingList. If the lists are not equal, *POModifyAck* message is discarded.

As system topology continues to be modified, the size of the POPendingList can get very large. To discard already processed changes to the PO Table, after receiving all *POModifyAck* messages, the root process sends out *POModifyDone* message. When the process receives this message, it subtracts the received POPendingList from its local list.

Finally, we must consider the case where the *POModifyDone* message does not reach every node due to another node failure. In this case the *POModifyNew* message will be sent to the root, and the root will rebroadcast *POModify* message. We have to make sure that no modification requests are processed twice. The process can not be removed from the PO Table twice. If the process is being added to the PO Table, we have to verify that the specified process is not already in the table before adding it.

The complete algorithm is presented below.

1. Message is sent to the root coordinator containing IP and PID of the process that is being added/removed. Root updates its PO Table, and, if the process is being added, it sends a copy of its PO Table to the new process. Then root adds the process to its POPendingList, and sends the *POModify* message to its children. This message includes local POPendingList.

   Until the update is finished, the Process Order is inconsistent, so root does not broadcast any application messages while updating Process Order is in progress.

2. Receiving *POModify* message: if local POPendingList is not the subset of the received one, send *POModifyNew* message to the root. This message includes local POPendingList.
   If local POPendingList is a subset of the received one, update PO Table. If children list changed, (as it did for the process *a* in figure 7), break connections with old children and establish connections with new children. Then forward *POModify* message to children.

   When a leaf process receives *POModify* message, it replies with *POModifyAck* message to its parent.

3. Receiving *POModifyAck* message: if local POPendingList is not the same as the received one, then discard the message. When the message is received with the matching POPendingList from all its children, send *POModifyAck* message to its parent.

   When root receives *POModifyAck* message from all its children, it sends *POModifyDone* message to its children. This message contains the root's POPendingList. The use of the hierarchy is resumed.

4. Receiving *POModifyDone* message: if local POPendingList is not the same as the received one, send *POModifyNew* message to the root. This message includes local POPendingList. Otherwise, clear local POPendingList and forward the message to children.

5. Receiving *POModifyNew* message: modify PO Table. If not root, then forward this message to the root. If root, modify local POPendingList to contain the union of local POPendingList and received POPendingList. Send *POModify* message to its children.

5 Process Order APIs

As discussed in the next section, there is often a need in distributed systems to construct structures other than binary trees. Process Order can be used to construct these structures. To facilitate easy use of the Process Order we are implementing it as a set Process Order library. Figure 8 presents a set of interfaces for this library. *POInit()* function initializes the Process Order Table, and inserts a host process in it.

Using the function *POAddCommPartners()* the user specifies a rule by which logical connections have to be established and then maintained. An example of such a rule is the one we used in section 3.2 to build a hierarchy. This function could be called multiple times with different rules, as Process Order can support multiple structures. Calling this function results in opening socket connections between processes that are according to the rules are logically connected.

Function *POGetCommPartners()* returns the set of the processes logically connected to the calling process according to the rule submitter in the function argument. If socket connections were established with *POAddCommPartners()*, then the corresponding socket descriptors are also returned.

POAdd() and *PORemove()* are used to add and to remove elements from the POTable. If *POAddCommPartners()* was called to establish socket connections on the logical links, then calling *POAdd()* and *PORemove()* will cause the creation and destruction of the socket connections according to the changes made to the logical connections. An example of such a reconnection was described in section 4.

To facilitate easy reconnection between processes in case of modification of the system topology, we use TCP/IP-based communication library developed by Christian Wicke [8].

6 Concluding Remarks

Aside from the hierarchy, it is often desirable to arrange processes in other logical structures, such as a ring [9], for fault detection, a hypercube [10], to implement stability detection algorithm, a mesh [11], for grid computations, or groups [12], for load balancing. Process Order could be used to construct and maintain these structures in a similar way as for the tree. The usability of the Process Order mechanism depends on how well the desired logical structure could be expressed with a formula.

```
POInit(key1, key2)
POAdd(key1, key2, process name)
PORemove(key1, key2)
POAddCommPartners( function )
POGetCommPartners( function )
```

Figure 8. Process Order API

The cost of maintaining the Process Order is a total of $3n$ 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 extra space: at least one integer per participating process. Note that the cost of maintaining Process Order is independent of how many structures are being concurrently supported by the mechanism.

As distributed systems become more complex, the need for supporting multiple logical structures will increase. The Process Order mechanism provides a set of

abstractions, that allow fast and clean creation and maintenance of such structures.

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] S. McCanne and S. Floyd. NS (Network Simulator) Home Page. http://www-nrg.ee.lbl.gov/ns

[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] C. Wicke. "Implementation of an Autonomous Agents System." Diploma thesis, University Karlsruhe, 1998

[9] E. Gendelman, L. F. Bic, M. Dillencourt. "Efficient Checkpointing Algorithm for Distributed Systems with Reliable Communnication Channels." TR #99-34 (Section 5.2), Department of Information and Computer Science, University of California, Irvine, 1999.

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

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

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