Pedagogical Tools for Distributed Systems

MASTERS PROJECT REPORT

presented to
Department of Computer Science
Michigan Technological University
in Partial Fulfillment of the Requirements
for the Degree of
MASTER OF SCIENCE IN COMPUTER SCIENCE

by

Meghana Gothe
msgothe@mtu.edu

MICHIGAN TECHNOLOGICAL UNIVERSITY
2000

This project, "Pedagogical Tools For Distributed Systems", is hereby approved in partial fulfillment of the requirements for the Degree of MASTER OF SCIENCE IN COMPUTER SCIENCE.

DEPARTMENT of Computer Science

Thesis Advisor Dr. Jean Mayo

Department Chair Dr. Linda Ott

Date

Acknowledgments

I would like to express my sincere gratitude to my advisor, Dr. Jean Mayo for her advice and suggestions and without whose support and encouragement, this project and report could never have been finished.

I also thank each of my committee members, Dr. Linda Ott, Dr. Adrian Sandu, and Dr. Mangalam Gopal(Mathematics Department), for their time and support. Special thanks to Munish Mehta for his guidance and constant encouragement.

Abstract

Teaching a distributed system class becomes a difficult task when the student community is from a varied background. These systems are complex, as they are distributed across the network of component processes. A majority of the students have a sequential programming background. Hence, distributed systems concepts pose a challenge of re-learning, making the teachers job all the more difficult. Furthermore, distributed systems rely on message passing for communication between component processes. The large amount of system specific network interface detail obscures the actual concepts. Tools that attempt to eliminate this obscurity are needed. The students are introduced to numerous protocols, which form the basic foundation of distributed systems programming. Shortage of time prevents students from implementing them all. A simulation of the protocols which can be used by the students for understanding is a need of time.

With this motivation, we have implemented a tool which provides certain communication primitives and some important distributed system protocols. The communication primitives will provide an easy interface that can be used by the students in their assignments. Protocol execution dumps will enable the students to correlate to the topics taught in class.

Contents

List of Figures

1 Introduction

A distributed system is a set of processes which do not share a common clock or common memory, and which communicate via message passing. The activity of each process in the computation is characterized as the execution of a sequence of events. The events are comprised of sends, receives, and internal events (events that are local to a process and change the state of that process). A system is said to be distributed if the message transmission delays are not negligible compared to the time interval between events in a single process. The ability to analyze the behavior of such a system is essential for various stages of its development like designing, testing, and debugging. The inherent complexity of distributed systems makes the above mentioned tasks difficult.

In general, the students taking the classes in distributed systems are from varied backgrounds and find it difficult to grasp the concepts of distributed systems. Programming assignments are the best means of making the students understand the important concepts of distributed systems. However, the large amount of detail needed for the underlying network interface routines obscures the conceptual understanding within the programming exercises. A set of communication routines that will ease this burden on the students is needed.

Distributed systems present to the students the challenge of *re-learning* many concepts. Simple issues like finding the state of a computation, enforcing mutual exclusion, and detecting termination are tremendously complicated. A set of routines implementing the protocols that address the issues within distributed systems will help the students understand, and facilitate the teaching process. However, in the short duration of an academic term, programming of all the different protocols is a very difficult task. The pedagogical tools system we have implemented will provide the students with routines that they can use to aid their learning.

Our project provides routines that implement four protocols : termination detection by Dijkstra,

et al. [2], Chandy and Lamport's distributed global snapshots [1], Byzantine Generals agreement by Lamport, et al. [5], and Ricart and Agrawala's mutual exclusion [10]. Along with these, we have implemented message passing primitives needed for the communication between processes executing the above protocols. The primitives support the following types of message passing: deterministic synchronous send-receive, deterministic asynchronous send-receive, nondeterministic synchronous receive, and nondeterministic asynchronous receive. We are also providing the ability to generate traces of the execution of the protocols, which will be handy for classroom discussions.

2 Related Work

Extensive research is being done in the field of distributed systems. The number of students taking courses in distributed systems and networking has increased. This makes it imperative to focus on work related to the development of tools to aid in their learning. The protocols and the message passing abstraction that we have provided form part of the basics of distributed computing concepts. Most of the currently available tools that attempt to provide similar implementations are customized for a particular application and cannot be incorporated into our pedagogical tool. Some tools of particular interest are :

- Didyma or Netsim [8] is a graphical network documentation and monitoring tool based on ICMP (ping) and TCP ports. The facilities the tool provides include: a graph of the status of the nodes in a network along with their history, raising of an alarm with a mail notification if an event (i.e. a change in status of a node) occurs, and output of the turnaround times in the network. One can also view the status of a network remotely with a browser using the Didyma HTTP service. However, this tool does not provide any functionality needed for the tool we are trying to provide.

- H.Jakiela describes the use of a simple visualization technique to resolve difficult queuing problems and system overload in a large distributed system [3]. Response time problems caused by lock contention, queuing delays, and soft faults can be easily identified with this technique. However, this tool also does not provide the protocols or the message passing routines that we are trying to provide in our tool.

- One of the tools which is similar to our needs is "DAJ: Toolkit for Distributed Algorithms Simulation in Java", written by Wolfgang Schreiner [12]. The toolkit implements some protocols, including two of the protocols needed by the project: termination detection and global snapshots. It also provides a graphical interface in Java for visualization of the protocol execution. An applet shows the simulation of the termination detection protocol on a network with six nodes. When a node is clicked, its state (active or passive), and its color are displayed. When a channel is clicked on, its state, either empty or non empty, is displayed.
  A simulation of taking a global snapshot is shown via an applet using a banking system example. When a node is clicked on, the node status in the form of the current balance and whether the node is taking a snapshot is displayed. The number of marker messages needed to be received by a node, and a snapvalue (value of the local balance after snapshot was initiated until all the markers are received) is also displayed. A constant number of nodes is used. The tool does not provide vector or Lamport logical timestamp information. Unreliable message passing is also not supported. Further, it does not implement all the selected protocols. Finally, the application could not be be used by our implementation because we have selected C++ for our tool because our tool is a part of a bigger "Concurrent Computing" project.

- Another tool, similar to the above and also in Java, is "Visualization of Randomized Distributed Algorithms", implemented by Minas Lamprou and Ioannis Psarakis. The package called "Simjava" [6] is used to simulate the execution of a set of randomized protocols such as Leader Election in a Ring, Dining Philosophers, and Byzantine Agreement. However, out of these, only the Byzantine Generals Agreement Protocol is of use for our purpose. The applet provided visualizes Ben-Or's asynchronous randomized distributed solution to the Byzantine Generals problem, in the case of one traitor and five loyal generals. We have implemented the agreement protocol proposed by Lamport, et.al. Simjava is implemented in Java, and hence does not suit our needs. Further, Simjava does not provide tools for unreliable message passing, or for developing user applications which use the protocols.

3 System Model

We consider a distributed system comprised of a collection of sequential processes $\{P_1, P_2, ..., P_n\}$, and a network capable of implementing unidirectional communication channels between pairs of processes for message exchanges. Channels may be reliable (no messages are lost in transit) or unreliable. Reliability is provided by the message passing primitives. The delivery of messages may be done out of order, i.e., two messages from a process may not be received in the same order they were sent. The communication network is assumed to be strongly connected (but not necessarily fully connected), i.e., every process can communicate with every other process directly or through intermediary processes. We assume that there exists no bound on the relative speeds of the processes, and message transmission delays are unpredictable. This system is realized in hardware by using a cluster of workstations.

4 Background Information

The following information provides some background in the form of definitions that are needed to understand the details and the importance of the message passing primitives and protocols that are to be implemented.

Consider a distributed system D as described in the previous section. Let E_i denote the set of events occurring in process P_i, and let $E = E_1 \bigcup ... \bigcup E_n$ denote the set of all events of the distributed computation. Here it is assumed that each P_i is sequential, and that the events in E_i are ordered by the sequence of their occurrence. Figure 1 shows a time-space diagram of a distributed system with three processes. The horizontal arrows represent increasing time and circular dots represent the events. The order of occurrence of the different events in the distributed system needs to be deduced. Lamport established the *"happened before"* relation to order local events of a process globally in a distributed system [4]. The *"happened before"* relation is denoted by the symbol "$\rightarrow$" in the of remainder of the document.

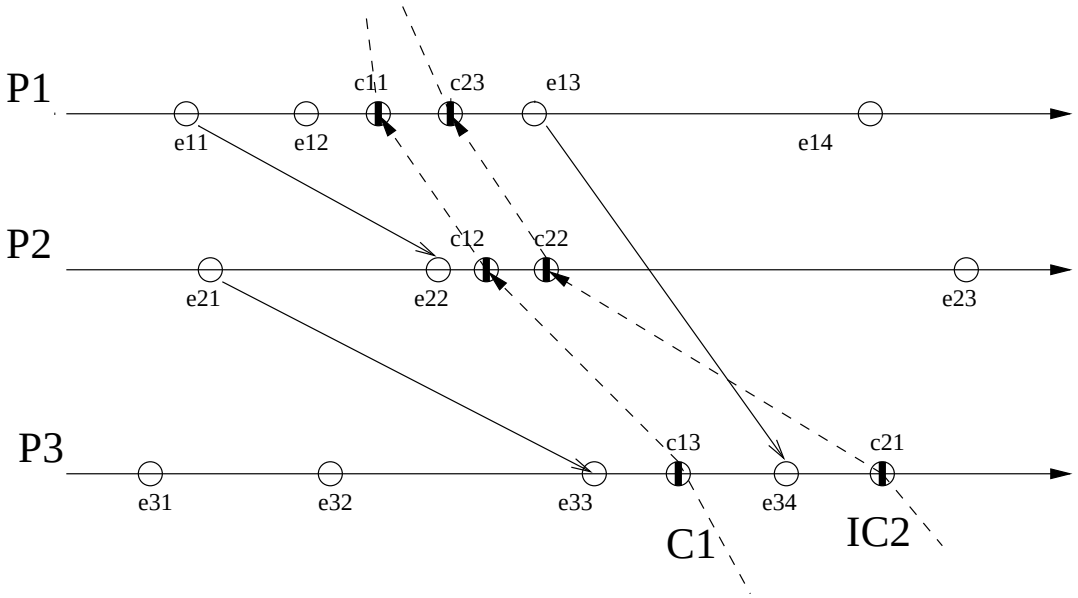

Figure 1: Time-Space Diagram of a Distributed System with 3 processes

4.1 Cuts

A *cut* C of an event set E is a finite subset $C \subseteq E$ such that $e \epsilon C$ and $e' \rightarrow e$, where e and e' are events in the same process, implies $e' \epsilon C$. A *cut event* c_i is an event in the cut that does not alter the state of the process. Figure 1 shows the time-space diagram of a distributed system. Cuts are shown by the dotted lines and the cut events are filled.

A *consistent cut* of an event set E is a finite subset $C \subseteq E$ such that $e \epsilon C$ and $e' \rightarrow e \Rightarrow e' \epsilon C$. There is no restriction that e and e' are in the same process [7]. In Figure 1, C1 gives a consistent cut, and IC2 is an inconsistent cut.

4.2 States

The *state of a process* is defined as the value of all variables used by the process. The *state of a channel* is defined as the sequence of messages sent along the channel excluding the messages received along the channel.

A global state of the system D is a set of local states, one from each process P_i, and the state of each channel C_i. The *initial global state* is one in which the state of each process is its initial state and the state of each channel is the empty sequence. The occurrence of an event may change the global state.

A fundamental property of the distributed system is the lack of a global system state. Some of the factors that prevent an observer from determining the global state are as follows.

1] A process can only keep track of the messages it sends or receives, and thus only its own state.

2] The systems lack a common clock.

3] Modern systems are complex with unpredictable problems like CPU contention, interrupts, and page faults.

4] In a distributed system, communication is not instantaneous due to propagation delays, con-

tention for network resources, and lost messages that require retransmission.

In most situations, instead of determining the exact global state of the system, a global view, called a global snapshot, that is consistent with causality is determined [9]. Events related by the happened before relation are said to be causally related, as one event may have caused the occurence of the other.

A *global snapshot* of the system is defined as a state of the distributed system that might have occurred [11]. A global snapshot is comprised of the set of process states at the time the cut events of a consistent cut are executed, as well as any outstanding messages in the comunication channels. Snapshots are useful for application monitoring and control, e.g. deadlock detection, and detection of token loss.

4.3 Stable Predicate

Let y be a predicate function defined on the global states of a distributed system D such that *y(S)* is true or false for a global state S of D. The predicate y is said to be a *stable predicate* of D if *y(S)* implies *y(S')* for all global states S' of D reachable from global state S of D. Thus, if y is a stable property, and y is true at a point in the computation of D, then y is true at all later points in that computation [1]. For example, if a system has terminated, the predicate corresponding to the detection of termination of the system will be true, and will remain true. Hence, the predicate is called a stable predicate. The processes in many distributed computations are considered to be comprised of a sequence of phases. Each phase is composed of an active part, where useful work is done, and a stable part, which indicates the end of a phase. This is analogous to a sequential program looping to produce some result, until successive iterations produce no change. Detection of stability is essential for terminating one phase and starting the next.

5 Project Work

This section describes the details of the project work, and the details about the message passing primitives and protocols that have been implemented. The project consists of the following parts which are described in the sections below.

1. Implementation of the control process and intialization routines for the remaining processes.
2. Implementation of message passing routines.
3. Implementation of the protocols.

5.1 Control Process and Initialization

1. *Implementation*

   As the first step of the project, a control process program has been implemented. This program performs important functions during initialization. The control process is bound to a port and it acts as a server. The objective of having such a process is to be able to start the distributed application from a single machine. The user can invoke this program from the command line, and will have to provide, as command line argument, a text file that will contain an integer identifier for each process, the name of the host machine where the process is to be executed, and the full pathname of the executables the user wants to execute on that host. The control process uses this data to spawn the appropriate component process on the machine indicated by the user. Every spawned process which executes the application specified by the user invokes an initialization routine called **myinit(myid, numprocs, argv, argc)**. Here, *myid* specifies the integer identifier of the process, *numprocs* specifies the number of processes involved, and *argc* and *argv* provide the process with the network address details of the control process. This routine binds the process to a port, and then forks a child

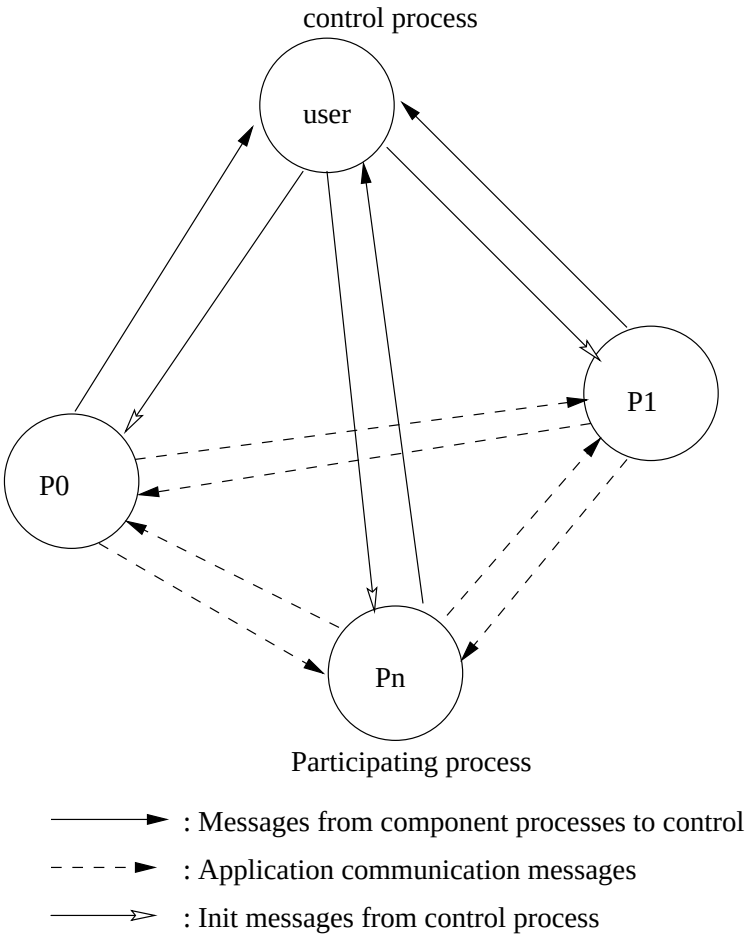

Figure 2: Proposed Model for Project Implementation

process, forming an underlying sublayer, with which it establishes a Unix based pipe interface. This layered architecture separates implementation specific details from the application layer, and allows ease in writing application programs based on simple API calls. The sublayer communicates with the control process in the following sequence.

- The network address details of the control process are obtained by each process during startup as arguments.
- The sublayer sends the local network address details, which include the process IP address and port number, to the control process.
- The control process collects the network address details of all the participating processes.

- The control process sends this information to each process.
- Every process stores this information in a special data structure from which the information can be retrieved whenever required.
- On receiving the collection of addresses and assimilating it, every process sends the control process an acknowledgement.
- After all such acknowledegments are received, the control process sends each process a message, which indicates that all the processes have been initialized and are ready to communicate.

A process in the computation can be terminated by invoking the routine **myclose()**, which informs the sublayer of its termination and closes the pipe interface to the sublayer to disallow further communication with the sublayer. After the computation is done, depending on the protocols used by the application, the user can view NFS mounted files that contain protocol specific execution details, which will be discussed in the following sections. The user can also view a file containing the dump of the vector clock and the Lamport logical timestamp information [11]. These files can be used for trace visualization of the execution of a distributed computation.

5.2 Message Passing

1. *Problem Description*

   Communication in a distributed application is provided by message passing. The basic message passing primitives are : *synchronous*, where blocking send and blocking receive are used for the communication, and *asynchronous*, where a non-blocking send and blocking receive are used. A blocking call requires the calling process to wait until either the communication

is completed or a failure is reported. A non-blocking call allows the application to proceed while the underlying communication sub-layer takes care of the communication. The message passing routines can be *deterministic*, where the sender and the receiver know the processes with which they will communicate, or *nondeterministic*, where the sender can send to anyone within a group of processes, and the receiver can receive from anyone within a group of senders. The nondeterminism makes the communication complicated, as the amount of overhead and the chances of error are increased. Problems like race conditions and deadlock can also occur.

The message passing primitives we implemented include: *Deterministic Synchronous Send-Receive*, *Deterministic Asynchronous Send-Receive*, *Nondeterministic Synchronous Receive*, and *Nondeterministic Asynchronous Receive*. The asynchronous receives are blocking in our implementation. Use of nondeterministic send is unusual, and hence is not being provided as a part of the project. The users have been provided with a set of routines which can be used to perform the desired communication. We are also providing the users an option to use reliable or unreliable message passing. Furthermore, users have access to Lamport logical time and vector time [11]. This will enable them to correlate to the classroom examples and generate execution traces when required.

2. *Implementing the Message Passing Primitives*

We have provided an abstraction layer on top of the original Unix communication routines using UDP/IP with the Berkley socket network interface. This involved writing new routines to carry out the functionalities of the message passing routines we are implementing. Both reliable and unreliable routines are provided. Reliability in the message passing is provided using acknowledgments, timeouts, and retransmission. Special primitives have been provided

to allow the user to change the reliability of a channel. Following is the description of available message passing routines.

i. **syncSend(receiver_id, datatype, count, message)** performs the synchronous send transfer of the message. A corresponding **syncRecv(sender_id, datatype, count, message)** routine, which performs the synchronous receive, is also provided. Here, *receiver_id* specifies the identifier of the receiving process. For nondeterministic receive, *sender_id* in the receive routine is set to negative one(-1). Currently, the data types supported are integer, character, float and double. On successful completion, the receive routine will return the identifier of the process with which the communication took place via *sender_id*, and the data received via the buffer pointed by message.

ii. **asyncSend(sender_id, datatype, count, message)** performs the asynchronous send of the message. This is a non-blocking routine. A corresponding **asyncReceive(sender_id, datatype, count, message)** routine is provided. In our implementation, the asynchronous receive routine is blocking. The parameters have the same functions as described above.

iii. A routine **ChangeReliability(channelId, value)** is provided that can be used to change the reliability of a channel. The user can specify the channel and a value between 0 and 100 for the probability of the delivery of a message along that channel, where 100 is the maximum probability of the message being sent. The default value is 100. This functionality is required, as the student applications are often designed for unreliable channels, but are generally executed on networks which have high reliability.

iv. The message passing routines implemented maintain vector and Lamport logical timestamp information. This vector and Lamport logical timestamp information is limited

currently to track only the sending and receipt of messages. This information is available to the user at the end of computation in a predefined file. The user is also provided with a routine **GetCurrentTimeStamps()**, to get the current information. This routine takes as an argument a character pointer in which the timestamps are returned to the application layer.

3. *Interaction with Sublayer*

The user can use the above routines to communicate with the remaining processes participating in the computation. When the user calls the send routines, the data is first packed into a character array. In the case of asynchronous send, the reliability of the channel is checked first. If the reliability is less than 100, the probability of sending the message is calculated by special functions: **CheckReliability(channelid)** and **flipCoin(reliabilityvalue)**, depending upon which the message is either dropped, i.e. control is returned back to the application, or further action is taken. Multithreading has been used to provide the asynchronous nature of the communication. Control is returned back to the application, while the thread carries out the remaining functions. In the case of synchronous send, multithreading is not used. The packed message is conveyed to the the sublayer along with the action to be taken on it. The sublayer attaches the necessary headers needed for reliable communication, such as the type of message, the sequence number and length of data. The new Lamport logical timestamp and the vector timestamps are calculated, and attached as a header. Acknowledgement and timeouts are used for reliable message passing. Success or failure is returned to the function in the application layer or the thread, which is handling the send.

When application data is received by the sublayer of a process, it is stored in a special queue structure by the sublayer. When the application requests data, the details are checked against

the messages already in the queue. If the data is available, the header is stripped off, necessary updates are made, and the actual data is delivered to the application. Otherwise, the application is made to wait until such a data is received. Before delivering data to the application, the data is converted to the data type needed by the application. Timeouts are used to prevent permanent blocking, and an error is returned to the application.

5.3 Basic Protocols

The following sections describe the basic protocols that are taught in the class, and our implementation that will aid in the students understanding of the subject. The objective of the design policy we followed was to provide a base on which additions could be done easily to extend the functionality of the tool. The implementation subsection for each protocol is itemized according to the sequence followed in the protocol execution, from the perspective of the user.

5.3.1 Global Snapshot

1. *Problem Description*

   A global snapshot is a state of the distributed system that could have occurred. Problems like stable property detection in distributed systems can be solved by using these snapshots. Examples of stable properties are "the computation has terminated", "the system is deadlocked", and "all tokens in the ring are lost". Global state thus obtained can also be used for checkpointing. Thus, understanding global states and the protocols devised for their detection are fundamental to reasoning about distributed systems. As mentioned in the previous sections, the task of taking the snapshot is complicated by the nature of distributed systems. Chandy and Lamport devised a simple means for taking a distributed snapshot [1].

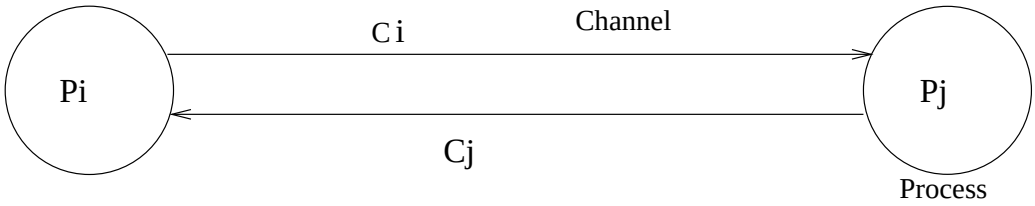

Figure 3: A Single Token Conservation System

2. *A Brief Description of Chandy, Lamport's Approach*

   The system consists of processes $P_1, ..., P_n$, connected by channels $C_1, ..., C_m$. Each process records only its own state, and the state of the incident communication channels, to form the *global state*. The channels are assumed to be unidirectional, reliable, and FIFO. The global state detection algorithm is to be executed concurrently with the user application, and is superimposed on the underlying computation without interfering with it. Chandy and Lamport used a *single token conservation system* to motivate their protocol, and provide a guideline for its implementation.

   If process P_i and process P_j are connected by channel C_i, the important restrictions in finding the states to obtain a global snapshot are outlined as follows.

   - If n is the number of messages sent along a channel C_i before process P_i's state is recorded, and n' is the number of messages sent along C_i before the state of C_i is recorded, then the recorded global state is consistent only if n **is equal to** n' .
   - Similarly, if m is the number of messages received along a channel C_i before the state of a process P_j is recorded and m' is the number of messages received along C_i before C_i's state is recorded, then the recorded global state is consistent only if m **is equal to** m'.
   - In every state, the number of messages received along a channel cannot exceed the number of messages sent along that channel, $n \geq m'$ **and** $n \geq m$.

So, the state of channel C_i that is recorded must be the sequence of messages sent along the channel before the sender's state is recorded, excluding the sequence of messages received along the channel before the receiver's state is recorded.

3. *Implementation*

The user is provided with routines which will enable him to take a snapshot of a distributed computation. Currently a single snapshot can be taken.

(i)The following two routines need to be called before calling the initialization routine **myinit()**. These enable the state of the process to be found dynamically.

- **InitProcessVar()** sets the pointers to variables that are used by the process. The value of these variables denote the state of the process. To record the process state, the user will have to make appropriate changes to the routine. The specific guidelines to do so have been provided in the source code for the routine.
- **SetSnapIndicator(value)** where value is a pointer to an integer that the user provides as an input to the routine. This acts as an indicator for the snapshot completion.

(ii)At some point in the computation, one of the processes can start taking the snapshot by invoking a routine **StartSnapshot()** provided for this task. Care has to be taken that only a single process makes such a call. The marker sending and receiving rules, described above, are followed while taking a snapshot.

(iii)The application is signaled asynchronously to get the process state and to indicate the completion of the snapshot using the indicator set by the routine **SetSnapIndicator()**.

(iv)The process and channel state values, along with the vector timestamps of the receipt of the markers, are dumped to a predefined file at the termination of the computation. These details will provide the user with the global snapshot, and also a provision to check the validity

of the snapshot.

4. *Interaction with the Sublayer*

We have provided a mechanism by which the snapshot can be taken dynamically and transparently to the application execution. When a process starts taking the snapshot by calling **StartSnapshot()**, it notifies the underlying layer to do so, and sends the process state to it. Markers are sent to all participating processes, and control is returned back to the application. When a marker is received by any process for the first time, the application is signaled asynchronously to obtain the process state. A dedicated pipe interface is used for exchange of data. Interrupt **SIGUSR1** is trapped for signaling purposes. The application is allowed to then perform normal execution. After the first marker, the sending and receiving of the markers is taken care of by the sublayer, transparent to the application. On the receipt of a marker, the sublayer records the state of the channel as the number of messages received along that channel after the first marker was received. It also records the vector timestamp of the marker receipt. After the markers are received on every channel, the application is signaled aynschronously regarding the completion of the snapshot using the same pipe interface and signal described above. At the termination of the distributed computation, every process dumps the state data into a predefined file, which provides the global snapshot.

5.3.2 Mutual Exclusion

1. *Problem Description*

In any multiprogramming environment where resources need to be shared, critical section processing is an important aspect. It is necessary to see that only one process accesses the resource at a time to avoid conflicts and corruption of data. Synchronization achieved by mutual exclusion attains utmost importance in distributed systems, as commonly many

processes are trying to access shared resources. Within single machine systems, mutual exclusion is achieved using semaphores and locks. However, since only message passing is used for information exchange, the techniques used in sequential programs are difficult to implement in distributed systems. We have implemented Ricart and Agrawala's algorithm [10]. The protocol is symmetric, as the same algorithm is executing on each node, and it requires no shared memory. This algorithm provides an optimal solution for the number of messages sent and uses only 2*(N- 1) messages between the nodes, where N is the number of nodes in the system. The previous similar work done for mutual exclusion by Lamport used 3*(N-1) messages per critical section invocation [4]. In addition, the time required to obtain mutual exclusion using Ricart and Agrawala's protocol is optimal. The underlying network is assumed to be error-free, and the nodes are assumed to operate correctly.

2. *Brief Description of Ricart and Agrawala's Approach*

A process which wants to enter the critical section sends a request containing the process identifier, and the timestamp for the request that is unique system-wide, to all the other participating processes. Upon receipt of the request, a process can reply immediately or defer the response until it leaves the critical section. Only after a process receives a reply from all processes can it enter the critical section. The request grant or deferment is determined using the unique timestamp, according to the priority of the incoming request, depending on whether the process has itself requested the critical section, and whether the process is itself in the critical section [10]. When a process exits the critical section, it informs every other process. Processes do not send an explicit denial message to the originator of the request, resulting in a reduction in the number of messages.

3. *Implementation*

We have provided routines that will enable a process executing the user application to achieve mutual exclusion for critical section processing or for using a resource.

(i) **GetCriticalSection()** is invoked when a process needs to execute in the critical section. The routine can be called after initialization. On invoking this routine, a request for critical section is sent to every process in the distributed application.

(ii) The application waits until control is returned to it by the sublayer. The application is blocked until then. Once the control is regained, the process starts executing in the critical section.

(iii) **ReleaseCriticalSection()** is called by the application process executing in the critical section, on completion of the critical section processing. This is to inform the participating processes of the completion of the critical section execution.

(iv) After the critical section is granted to a process, the identifier of the process granted the critical section, and the request number, are written to a predefined file. If the critical sections are granted according to the algorithm, the file will contain the data written by the processes in the order in which they were granted the critical sections, or were using the resource. The contents of the file can be examined to check the sequence in which the processes obtained the critical section. Thus, the user can understand how mutual exclusion is achieved, as well as the issues like message passing overhead that are related to achieving synchronization

4. *Interaction with the Sublayer*

The sublayer design of the project enables the protocol to be invoked by simple API calls, and execute transparent to the application layer. Whenever a process participating in the distributed computation wishes to execute in the critical section, the application sends a

command to the sublayer. It waits until the critical section is granted. The sublayer intializes the data structures to maintain a record of the replies received, and the requests that might be deferred. The unique timestamp is deduced using a variable initialized at startup, and updated at every request sent and obtained. The request is sent to all the processes using this sequence number. Whenever a reply is received by the sublayer, it checks to see if it is a duplicate. Otherwise the reply count is incremented, and the necessary data structures are updated. After all the replies are obtained, the sublayer writes the request number and process identifier to the designated file, and sends a success value to the application. The process can then execute in the critical section. If a request is received while the process is requesting the critical section, or is executing in the critical section, then the appropriate decision to defer the request is taken by the sublayer, depending on the request number and the process identifier number. A record of any deferred requests is kept in a special data structure. If the process is neither in the critical section nor is requesting the critical section, a reply is sent immediately to the requesting process. Once the process finishes executing in the critical section, it sends a command to the sublayer to notify the remaining processes of this change. The sublayer then sends a reply to any process that it had deferred a reply to, and returns control back to the application. Any protocol related messages from other processes are received and handled only by the sublayer. This provides the necessary abstraction needed for the transparent execution of the application.

5.3.3 Termination Detection

1. *Problem Description*

   In certain distributed computations, a process is considered to be in one of two states: active, in which it is doing some useful computation, or passive, in which it is either idle or has

terminated. A process becomes passive on completion of its task. Only active processes can send application messages. The receipt of an application message triggers a passive node to transition to the active state. The state in which all nodes are passive and no application messages are in the channels is stable, and the distributed computation is said to have terminated. The lack of a common clock, and the difficulty in determining the global state of a distributed application, make it difficult to detect the termination of distributed and parallel computations. In sequential programs, termination detection is an unheard of problem. The termination detection problem is well known in the field of parallel programming as the *barrier synchronization* problem.

The problem of termination detection is a good introduction to the topic of distributed systems. It is simple and illustrates the problems of distributed systems programming. We have implemented the distributed termination detection protocol developed by Dijkstra, along with Feijen and Gasteren, which uses synchronous message passing [2]. The solution provided by them is simple, elegant, and easily understood.

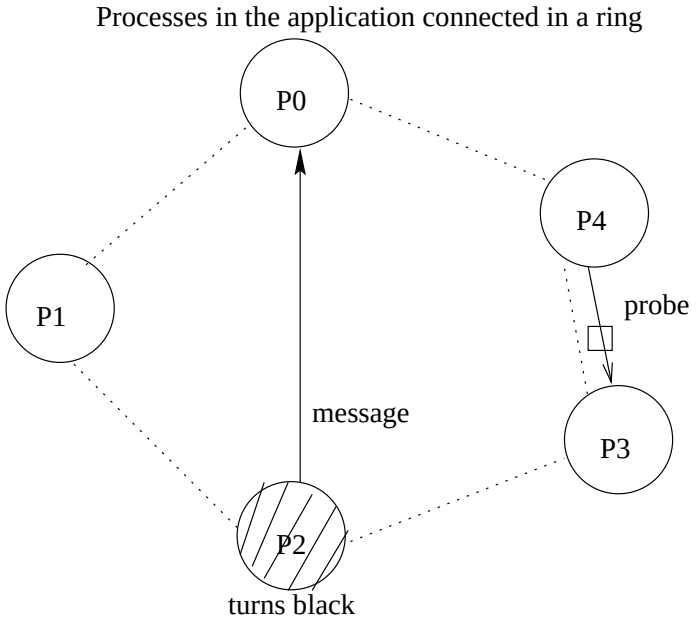

Figure 4: An Example For Termination Detection Algorithm

2. *Brief Description of the Approach and the Invariants*

For the purpose of transmitting messages associated with the detection of termination, processes are connected in a ring fashion as shown in Figure 4. Initially all processes are passive; when the computation is initiated, the processes become active. The termination detection protocol is initiated by process P_0 using a special message called a 'probe', which it sends to process P_{N-1}, where N is the number of processes in the system. When a process P_i, $i > 0$ receives the probe, P_i propagates the probe to process P_{i-1}, after P_i becomes passive. Process P_0 transmits the probe to P_{n-1}. A passive process that has propagated the probe could become active when it receives an application message. To accomodate this fact, the probe and each process are labeled either white or black. When P_i is black, this indicates that P_i has sent a task message to a process P_j, where $j > i$. P_0 initiates termination detection by sending a white probe and making itself white. If a process P_i that holds the token is black, P_i will set the probe color to black before sending the probe to process P_{i-1}. If P_i is white, the probe is passed unchanged. After P_i passes the probe to P_{i-1}, P_i becomes white. A black probe returned to process P_0 indicates that the probe must be circulated once more, as some of the processes may still be active. In this case, P_0 will first make itself white, and then transmit another white probe. Upon receipt of a white probe, a white P_0 will declare termination.

3. *Implementation*

The implementaion provides the user with routines that can be used to start the detection of termination of a distributed computation. The termination detection protocol is superimposed on the application.

(i) **StartTermDetect()** is called by the process P_0 to start the termination detection pro-

tocol. This results in the transmission of the probe message as described in the previous section.

(ii)**ChangeMyState(state)** is invoked by a process to change its state to passive. This routine informs the sublayer of the change in state.

(iii)The routine **myclose()** is called by the application layer of a process to inform the sublayer that the processing has been completed. The application layer is then terminated. The connection to the sublayer is closed to prevent any kind of communication between the two layers.

(iv) When termination of the distributed system is finally detected, each process dumps the sequence of the recorded process and probe colors to a predefined file. This file will provide the students a good guide of how the detection algorithm was executed and the sequence of color changes that occurred during the execution.

(v) If a process is passive when termination is detected according to the protocol description, the application layer is terminated by a **SIGKILL** signal.

(vi) Our tool will be running distributed computations. There was a need to detect the termination of any such computation. The termination detection protocol described above was used for this purpose. It is started by process P_0 after it gets a message from the control process that all the processes have been initialized.

4. *Interaction with the sublayer*

Once process P_0 receives the message from the control process that all the processes have been initialized, P_0 invokes the routine **StartTermDetect()** to inform its sublayer to transmit a white probe, and change the process color to white. The protocol execution is thus started. During the computation, if the routine to change the state of the process from active to passive

is invoked, the sublayer makes a note of this change. When a data message is received, the sublayer changes the state of a passive process to active, while the state of an active process is left unchanged. It also makes a note of the state change when it is informed of the termination of the application process. Every time a data message is sent by the process, the sublayer changes the color of the process to black. When the probe is received by the sublayer, it first checks the state of the process. If active, the sublayer keeps the probe until the application layer notifies the sublayer of a change in process state to either passive or terminated. If the process is passive or has terminated, the sublayer first changes the color of the probe according to the rules [2], transmits the probe, and then changes the process color to white. The sublayer keeps track of the process color changes, as well as the received and sent probe color. The routines used for keeping track of the probe and process colors are **SetProbeColor()**, **GetProbeColor()**, **SetProcessColor()** and **GetProcessColor()** respectively. Process P_0 detects the termination of the distributed computation as described previously. If termination is not detected, P_0 retransmits a white probe, after changing its color to white. When termination is finally detected by process P_0, it sends a special message to all the processes. On receiving this message, the sublayer of each process calls a routine **DoFileWriting()**, which in turn invokes routines to write the timestamp details, the probe-process color changes, and any other protocol specific data to files, after which the child exits. If termination is detected when the process is passive, the sublayer sends a **SIGKILL** signal to the application layer and closes the pipe interface to prevent any message exchange.

5.3.4 The Byzantine Generals Problem and Agreement Protocol

1. *Byzantine Generals Problem*

The Byzantine army had to decide on an attack plan during a war. The generals could com-

municate using only messengers. The messengers were reliable. After observing the enemy, the generals had to decide upon a common plan of action. However, a small number of traitor generals could prevent the remaining generals from reaching a majority decision. This could be done by corrupting the data they sent to other generals, or by not sending any data at all [9]. Hence, the problem boils down to ensuring the following *interactive consistency conditions*:

IC1 All loyal generals will agree on the same value.

IC2 If the commanding general is loyal, the loyal generals will agree on his value and obey the order he sent.

Applying the situation to distributed systems, the Byzantine's problem becomes one of achieving a consensus. Processes begin with some initial value and must agree on the same output value, despite failures. Due to failures, inputs to processes may be arbitrary. Reaching a consensus is difficult in a distributed system, because communication is only through message passing, and it is difficult to detect correctness of an input. Consistency validation of a value is needed as some process may be faulty, making it difficult to determine if the value received is uncorrupted. This is a simple understanding of the consensus problem. The ability of obtaining consensus despite failures is required in many applications, like fault-tolerant clock synchronization. Other examples are maintaining replicated data, monitoring distributed computations, and detecting faulty processes.

We have implemented the solution to the Byzantine Generals problem proposed by Lamport, Shostak, and Pease, called the *Oral Message Algorithm* [5]. They proved that an agreement cannot be reached if $M \leq 3t$, where M is the total number of generals, and t is the number of traitors.

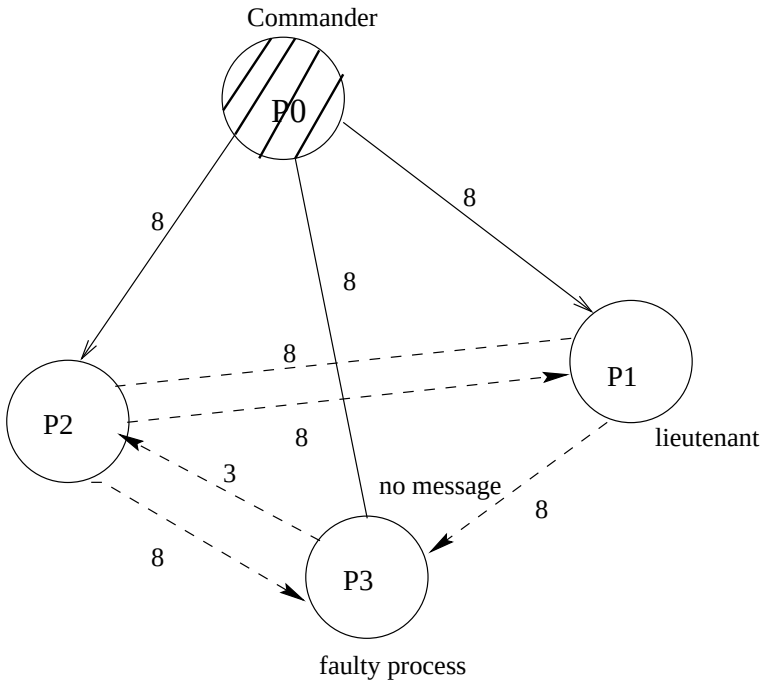

Figure 5: An Example For Byzantine Generals Protocol Algorithm

2. *Some Considerations for the Algorithm*

A distributed system consisting of N processes connected by channels is considered. The following assumptions are made.

1] Message delivery is reliable.

2] When a message is received, its sender can be determined reliably.

3] If a message is not delivered, its absence can be detected.

4] The default value is zero.

5] There are no spurious messages.

We denote the oral message solution by BG(k) [9]. Figure 6 outlines the algorithm used for reaching the agreement. Here, BG_Send(k, v, l) denotes broadcast by a process, and BG_Receive(k) denotes receipt of the value by a process, where k is the number of faulty processes that can be tolerated, v is the private value to be broadcast or rebroadcast by a process, and l is list of processes who have never broadcast the value v.

```
BASE CASE:
  BG_Send(0,v,l)
    The commander broadcasts "v" to every lieutenant in "l"
  BG_Receive(0)
    Use the value sent by the commander, or zero if no message is received.

BG_Send(k,v,l)
    Send the value "v" to every process in the list "l".
BG_Receive(k)
    1] Use the value "v" sent by the commander, or zero if no value is received.
    2] BG_Send(k−1,v,l−P_i ), where P_i denotes the process invoking BG_Send.
    3] Use BG_Receive(k−1) to receive v(i) for every process in l−P_i
    4] Return majority (v, v(1), ..., v(l−1)).
 majority (v1, v2, ...,vn) demotes the method that returns the majority value "v"
   among v1,v2,...,vn, or 0 if no majority exists.
```

Figure 6: Algorithm For Byzantine Generals Agreement Protocol

The BG(k) protocol is executed by each lieutenant process P_i recursively [5]. The remaining processes $P_j \; \epsilon \; |l - P_i|$, form a list of values they received from every other P_j regarding the value P_i received from the general. A majority regarding the value P_i received from the commander is found and stored in a list of such majority values. When the values of all lieutenant processes are obtained, a final majority value is found. Whenever a majority does not exist, a default value is used.

3. *Implementation*

The layered architecture we have implemented allows the concurrent execution of the protocol.

(i) **SetDecisionIndicator(value)** is called by every process executing the user application before invoking **myinit()**, if the protocol is to be executed. Value is a pointer to an integer variable which the application uses to obtain the agreed upon value. This routine initializes the necessary interface and variables needed for the protocol execution.

(ii) The **BGSend(value)** routine that we have provided the user can be invoked on the process from which the user wants to begin the agreement protocol. The routine will cause

the broadcast of the value of the private variable obtained as a parameter to all the processes involved in the computation. Currently only integer values are being handled.

(iii) The application is signalled asynchronously regarding the completion of the agreement process, and the value agreed upon is also sent to it. The default value when no majority exists is zero.

(iv) The list of majority values, the final majority decision, and the identifier of the process making that decision are printed to a file. The file thus describes what value each process received, and how it arrived at the decision. The file will help the students understand the execution of the agreement protocol.

4. *Interaction with Sublayer*

The user invokes the agreement protocol for a private value of a process by calling the routine **BGSend(value)**. This routine commands the sublayer to broadcast the value to all the participating processes. On receiving an agreement protocol message, the sublayer invokes **BGReceive(newMessage)**. The receive procedure checks whether the message is from the intiating process or from any other process, and takes the action accordingly. Each process broadcasts its version of the intiator's value. Once all such values are received, depending on the number of processes, either the final decision is taken, or the values received from other processes are broadcast on a round basis, where the round number is the process identifier number being currently handled. These broadcasts are made using the procedure **BGSend(processid handled, value)**. The values received are collected, and the majority for each process is found using the procedure **FindMajority()**. After all such majority values are obtained, a final majority is calculated. The sublayer then signals the application that the agreement has been reached, and provides the application layer the value which is stored by

the signal handler routine in the pointer previously provided by **SetDecisionIndicator()**. The interrupt **SIGUSR2** and a dedicated pipe interface have been used for this purpose. Before sending the signal, the final majority value as well as the array of values used to calculate this majority along with the process identifier are printed to a file.

6 Future Work

We have tried to implement a comprehensive system that can be used for course study. However, due to time constraints we could not implement certain features that will increase the utility of this tool. The various enhancements that can be implemented are described below.

- Add other protocols to make a full fledged distributed systems pedagogical tool software.
- The implementation of Chandy, Lamport's snapshot protocol can be extended to make provision for multiple snapshots.
- The implementation for Byzantine Generals agreement protocol currently deals with integer values and can be extended with some modifications to handle other data types also.
- We have currently implemented the Byzantine Generals algorithm with the cases of four and seven processes. These cases are the ones which are covered in class. The code can be extended to cover more cases, if need be, with some modifications.
- The output results sent to the files can be displayed in a GUI format for easier interpretation.
- Time-space diagrams can be created using the Lamport logical and vector timestamp values that are stored to file.

7 Conclusions

With the increasing use of networks, distributed applications are gaining importance. The student community which takes courses in distributed systems is from a varied background. Many find it difficult to understand the implications and issues faced while implementing such applications. The protocols and the message passing routines we have implemented as a part of our tool form a basic foundation for the understanding of distributed computations. This project will provide a very useful tool, not only for the professor to teach, but also for the students use to help in their understanding of distributed system concepts. It will enable the students to visualize the issues related to distributed application programming. The protocol execution dumps and the timing information will help them correlate to the material covered in class. Thus, we conclude that the tool we have implemented will be highly useful for distributed system course study.

References

[1] K.M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[2] E. Dijkstra, W. Feijen, and A. van Gasteren. Derivation of a termination detection algorithm for distributed computations. *Information Processing Letters*, 16:217–219, 1983.

[3] H.Jakiela. Performance visualization of a distributed system: A case study. *IEEE Transactions on Computers*, 28(11):30–36, 1995.

[4] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[5] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals problem. *ACM Transactions on Programming Languages and Systems*, 4:382–401, 1982.

[6] Minas Lamprou and Ioannis Psarakis. *Visualization of Randomized Distributed Algorithms, Simjava.* http://www.cs.bham.ac.uk/teaching/examples/simjava/, 2000.

[7] Friedemann Mattern. Virtual time and global states of distributed systems. In M. Cosnard et. al., editor, *Parallel and Distributed Algorithms: Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B. V., 1989.

[8] Parsis network tools. *Didyma.* http://www.xs4all.nl/ houtriet/, 2000.

[9] T.Johnson R.Chow. *Distributed Operating Systems and Algorithms.* Addison-Wesley, 1997.

[10] G. Ricart and A.K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, 1981.

[11] R.Schwarz and F.Mattern. Detecting a causal relationships in distributed computations: In search of the holy grail. *Communications of the ACM*, 19(5):279–285, 1976.

[12] Wolfgang Schreiner. *DAJ: Toolkit for Distributed Algorithms Simulation in Java.* http://www.risc.uni-linz.ac.at/software/daj/, 2000.