Sharing memory in distributed systems

Oscar Rodrigo Aguilar
University of Nevada, Las Vegas

Follow this and additional works at: https://digitalscholarship.unlv.edu/rtds

Repository Citation
https://digitalscholarship.unlv.edu/rtds/92

This Thesis is brought to you for free and open access by Digital Scholarship@UNLV. It has been accepted for inclusion in UNLV Retrospective Theses & Dissertations by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.
INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Sharing memory in distributed systems

Aguilar, Oscar Rodrigo, M.S.
University of Nevada, Las Vegas, 1990
SHARING MEMORY IN DISTRIBUTED SYSTEMS

by

Oscar Rodrigo Aguilar

A thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science
in
Computer Science

Department of Computer Science
University of Nevada, Las Vegas
August 1990
The thesis of Oscar Rodrigo Aguilar for the degree of Master of Science in Computer Science is approved.

Chairperson, Ajoy Kumar Datta, Ph.D

Examining Committee Member, Laxmi Gewali, Ph.D

Examining Committee Member, Kazem Taghva, Ph.D

Graduate Faculty Representative, Ebrahim Salehi, Ph.D

Graduate Dean, Ronald W. Smith, Ph.D

University of Nevada, Las Vegas

August, 1990
ABSTRACT

A set of processes sharing a block of common memory is one of the most frequent configurations for distributed systems, and is called the shared memory model. We will study this model and focus on the techniques for the specification of concurrent data objects and the verification of their implementation. We will also deal with the development of algorithm translations between the shared memory model and the message passing model, another highly frequent configuration in distributed systems where the processes communicate by exchanging messages only. The translation of algorithms between these two models is based on the fact that it is possible to simulate shared memory synchronization primitives in message passing systems. We propose an algorithm for simulating atomic registers, test-and-set, fetch-and-add, and read-modify-write registers in a message passing system. The algorithm is fault tolerant and works correctly in presence of up to \( \lceil N/2 \rceil - 1 \) node failures where \( N \) is the number of processors in the system. The high resilience of the algorithm is obtained by using randomized consensus algorithms and a robust communication primitive. The use of this primitive allows a processor to exchange local information with a majority of processors in a consistent way, and therefore to take decisions safely. The simulator makes it possible to translate algorithms for the shared memory model to that for the message passing model. With some minor modifications the algorithm can be used to robustly simulate shared queues, shared stacks, etc.
ACKNOWLEDGEMENTS

I dedicate this thesis to my parents for their encouragement, support, and their always opportune advice. They are the source of my inspiration and strength. I would like to express my gratitude to my advisor Dr. Ajoy Datta for introducing me to the area of Distributed Systems, for his excellent advice, suggestions, and for many interesting discussions that helped to improve the quality of my work. I would also like to extend my appreciation to Dr. Evangelos Yfantis who was my initial advisor when I first arrived to UNLV. Acknowledgement is also due to three other members of my thesis committee, Dr. Laxmi Gewali, Dr. Kazem Taghva, and Dr. Ebrahim Salehi.
## Contents

1 INTRODUCTION

1.1 Properties of the Distributed Computer System ........................................... 13

1.2 Advantages and Disadvantages of Distributed Systems .............................. 14

1.2.1 Advantages ................................................................................................. 14

1.2.2 Disadvantages ............................................................................................ 16

1.3 Architectural Models for Distributed Systems ............................................. 17

1.3.1 Workstation/Server Model ......................................................................... 18

1.3.2 Processor Pool Model .................................................................................. 19

1.3.3 Integrated Model .......................................................................................... 20

1.3.4 Comparing Different Architectures ............................................................. 20

1.4 Distributed Operating Systems ................................................................. 21

2 CONCURRENT OBJECTS AND CONCURRENT SYSTEMS ......................... 26

2.1 Some Definitions for Concurrent Systems ................................................... 27

2.2 Specification of objects .................................................................................. 31

2.3 Register axioms ............................................................................................. 32

2.3.1 The Axioms ................................................................................................. 38

3 ATOMIC REGISTERS .................................................................................. 39
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2.2</td>
<td>Implementation of the procedure \textit{EXCHANGE}</td>
<td>91</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Implementation of \textit{BITCONSENSUS} and \textit{FLIP.GLOBAL.COIN}</td>
<td>93</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Results</td>
<td>93</td>
</tr>
</tbody>
</table>

6 \textbf{SUMMARY}                                                                 | 95   |
List of Figures

1.1 A typical Distributed Computer System .......................................................... 11
1.2 The two main layers of a Distributed Operating System ............................... 22
1.3 The Amoeba Architecture .................................................................................. 24
1.4 A typical V configuration ................................................................................... 25

2.1 The sequence of read and write operations by the processes $P_1$ and $P_2$ ...... 29
2.2 The operations by the three processes on a register. It is possible to rearrange
the operations in such a way (as shown in Figure 7) that the nonconcurrent
operations preserve their order and the read operations return values that are
consistent with the write operations. ................................................................. 33
2.3 The operations of Figure 6 rearranged in time to return consistent values ...... 35
2.4 The operations by three processes on a register. These operations cannot be
rearranged in time to return consistent values because $Rd_1(1)$ cannot precede
$Wr_3(0)$ ........................................................................................................... 36

3.1 A reader process and a writer process accessing a shared register. The type of the
register determines which values are legal for a read operation to return when it
overlaps in time a write operation ....................................................................... 41
3.2 (a) Concurrent read and write operations executed non-atomically. (b), (c) Two different ways to execute the operations of part (a) atomically .............................. 48

3.3 (a) Two consecutive read operations to a shared register overlapping in time with a write operation. (b) (c) (d) Three different ways of simulating serial execution of the operations in part (a). ............................................................... 50
Chapter 1

INTRODUCTION

A distributed computer system consists of a set of autonomous computers linked by a network. It is designed to enable the individual computers to share resources in the network, such that they provide the same computing facilities to geographically dispersed group of persons or institutions. Users of a distributed system are given the impression that they are using a single, integrated computing facility, although the facility is actually provided by more than one computer and the computers may be in different locations. The most important characteristic of a properly designed distributed system is transparency. Transparency offers to the user a unified interface to a networked collection of computer systems, providing access to programs and data objects located in any of the computers in the network using the same name and operations regardless of their location. The motivations for building distributed systems became apparent with the development of single user workstations, servers, and high-speed local networks in the period 1971-1980. Time sharing centralized systems of that time had not shown a good performance when running highly interactive tasks such as graphics applications. In these cases single user systems had shown superiority due to the dedicated processor power that enabled application programs to maintain an interactive dialogue with the user without interruption. Also,
the direct connection of the display screen to memory enabled programs to display and modify information on the screen almost instantaneously. The first workstation developed was the Alto, designed at Xerox PARC between 1971-1973. Later in the eighties with the development of fast and low cost 16-bit and 32-bit microprocessors, other companies like Sun Microsystems and Apollo developed high speed and high performance workstations. Examples of them are the Sun-2 and Sun-3 workstations and the Apollo Domain DN300 and DN600 workstations. These workstations have speeds in the range of 1 to 3 MIPS. Figure 1.1 illustrates the different components of a distributed system.

![Diagram of a typical distributed computer system](image)

Figure 1.1: A typical Distributed Computer System

The initial motivation for building distributed systems was the necessity to share resources.
For example, program development requires the sharing of source and object code of programs. Office applications require the sharing of documents and other files. The necessity of sharing information originated the development of file services. A file service provided processes in other computers in the network (the clients) with facilities to store and access data files in a manner similar to the filing systems of most conventional operating systems. Processes in different computers could share files and most of the times specialized computers called file servers were used to store the files and to provide access and concurrency control. Access control is used to ensure that only authorized users could access the files. Concurrency control measures are used to ensure that updates by different clients to the same file were properly sequenced. The Xerox Distributed File Service is an example of such file services. Other distributed systems developed between 1970 and 1980 are the Cambridge Distributed Computing System, the Apollo Domain System, the Newcastle Connection, and Locus. In the eighties there has been a rapid expansion of research and development activity in the area of Distributed Systems. Among the most outstanding and successful research projects are Accent, Mach, Amoeba, Argus, V-system and Chorus. Parallely, many distributed systems currently in use have evolved in an environment consisting of a multi-user computers and workstations with traditional operating systems such as UNIX linked by a network. A common practice is to have a network of multi-user computers and workstations running BSD 4.2 UNIX with network file service software such as the Sun Network File System (NFS). The BSD 4.2 version of UNIX provides the conventional UNIX operating system in each separate computer with its own hierarchical file naming scheme and password file. The Sun Network File System allows any computer in the network to export the names of any of its file stores, allowing them to be mapped as a part of the file name space in other computers. The actual mapping is accomplished by an extended version of the UNIX mount operation, allowing a remote file system to appear as a part of the directory hierarchy in
the machine performing the mount operation. The network file system software in the computer that has mounted remote files stores intercepts the read, write and other file operations that refer the remote files and map these references to the correct files in the remote computer. It is worth mentioning at this time that these UNIX based systems are not considered to be completely distributed because they lack some of the properties of what is considered a truly distributed systems. The discussion of these properties is the topic of the next section.

1.1 Properties of the Distributed Computer System

The fundamental properties of a Distributed System are fault tolerance, transparency, and the possibility to use parallelism. A distributed system must be able to continue working in the face of single-point failures, and must be capable of parallel execution. So it must have the following:

1. Multiple processing elements that can run independently, each processing element or node must contain at least a CPU and memory. These processing elements must fail independently such that the failure of one node does cause the failure of the whole system.

2. Interconnection hardware that allows processes running in parallel to communicate and synchronize. The interconnections must be reliable too. The processors that are working correctly must always be able to coordinate their work.

3. Shared state for the distributed system. In this way a node failure does not cause part of the system state to be lost.

In addition file transparency should be implemented such that it provides:

1. Location transparency. File names may be used without knowing the location of files.
2. Concurrency transparency. Facilities must be provided such that several users may use the same file or the same group of files at the same time without corrupting them.

3. Failure transparency. Even if a client or server crashes during operations on files or group of files, the files must remain in a consistent state.

Systems built based on BSD 4.2 UNIX and Sun NFS do not have the properties of concurrency transparency, shared state and failure independence, this is why they are not considered completely distributed.

1.2 Advantages and Disadvantages of Distributed Systems

Distributed Systems often evolve from networks of workstations. Owners of workstations connect their systems together because of the desire to communicate and share information and resources. Information is generated in one place and often needed in another. People and information are usually geographically distributed. For these situations, the use of distributed computer systems is the most appropriate.

1.2.1 Advantages

The following are the advantages that users and system managers can expect to derive from the replacement of conventional systems by distributed ones:

1. Predictable response. Distributed system have a very good performance and time response when used for highly interactive applications or applications that require significant processing capacity. The dedication of the workstation processing power to support a single user, ensures a rapid response.
2. Cost. The cost of a computer depends on its performance and the amount of memory it has. The price of processors and memory and consequently the price of workstations is going down very fast. The cost of communication depends on the bandwidth of the communication channel and the length of the channel. Bandwidth may be increased, but not beyond the limits set by cables or interfaces used. The cost of replacing cables to increase bandwidth, specially in the case of wide area networks is high. As man-machine interfaces become more interactive, users want instant visual or audible feedback from their user interface. Latency caused by distances of more than a few kilometers of network is too high most of the times. Centralized computers that give the required number of cycles for these applications and the cost of the network technology that gets the required number of bits out to the users screen is prohibitive. Also computers organized in a distributed system may share expensive resources such as high quality laser printers and high capacity storage devices.

3. Extensibility. Distributed systems are built in a modular fashion since they are composed by autonomous computers. They are capable of incremental grow as the demand for service increases. The addition of new file servers or computers can be done easily without replacing existing interfaces. The limiting parameter is in any case the network bandwidth since each active workstation adds to the communication load in the network. The capacity of any centralized system, on the other hand, is the one that imposes limits on the maximum size the system can grow.

4. Availability and reliability. In a distributed system data is often replicated. A distributed system usually has built-in redundancy on the resources that can fail to provide fault tolerance. Therefore, distributed systems have the potential to be available even when arbitrary single point failures occur.
1.2.2 Disadvantages

1. Loss of flexibility in the allocation of memory and processing resources. In a centralized computer system or a tightly coupled multiprocessor system all of the processor and memory resources are available for allocation by the operating system according to the work to execute. In a distributed system, the largest task that can be executed is determined by the processor and memory capacity of the workstation used.

2. Dependence on network performance and reliability. Failure of the network causes the service to users to be interrupted, or the execution of tasks that require internode coordination to be aborted. Overloading the network degrades the performance and responsiveness to the users.

3. Security weaknesses. To achieve extensibility the software interface is made available to the users. Clients have access to the communication software and can access the interface to the servers. This creates the necessity of having security software to protect the services against accidental or intentional violation of access control or privacy constraints.

The Design Complexity of Distributed Systems

The design of distributed systems is complex mainly because the interconnection of well understood components often generates new problems not apparent in the components. Most of this complexity is apparent through the unexpected behavior of systems that are believed to be understood. In some cases, formal methods can be used to help predict what will happen when two systems are interconnected. Examples of problems that occur in the design of distributed systems are:

- Interconnection problems. Different computers may have different ways to represent data or provide services such as electronic mail or file services.
• Interference. Concurrent access to the same file, or to the network channels are examples of interference that causes problems in a distributed system. Special software has to be designed to handle cases of interference.

• Propagation of effect. Failures in one component of the interconnection network can bring the system down if the design is not careful. Potential bottlenecks must be identified and eliminated. Also design effort must be devoted to localize the effect of failures as much as possible.

• Effects of scale. In some cases a resource does not scale up with the rest of the components and becomes a bottleneck. This is usually the case of the network bandwidth.

• Partial failures. A distributed system must continue to operate in presence of partial failure of the components. When a node fails its behavior may become erratic, the system must be able to hide the effect of this erratic behavior it and proceed with its work normally.

Distributed systems are complex because what they have to do is complex. Services such as authentication, access control, maintaining quota, concurrency control to handle concurrent reads and writes to files, recovery mechanisms, among others are not easy to implement. Sometimes simple solutions cannot be implemented because they are too expensive. For example, a simple way to achieve a reliable network that stands several link failures without becoming partitioned is to design it as a fully interconnected network with one physical link between every two nodes in the network. The implementation of such network is extremely expensive.

1.3 Architectural Models for Distributed Systems

The architecture of a Distributed System identifies the main hardware and software components and modules of the system and defines the relationship between them. Important aspects of this
are the types of computers used, their location in the network, and the locations at which system programs and application programs are executed. We defined a distributed computer system as being composed of a group of autonomous computers linked by a network. Distributed systems of this type are also known as *loosely coupled systems*. There is another multiprocessor scheme that is also considered by people as distributed. This scheme consists of a set of processors sharing a single memory or address space. Distributed systems of this type are known as *tightly coupled systems*. In *tightly coupled* multiprocessor systems all the hardware resources are under the control of the same operating system. The operating system allocates processors and memory space to user's tasks and lets them run concurrently. The use of shared memory allows the user's tasks to communicate with each other and with the operating system through the use of shared variables. In this type of systems the maximum number of processors that can be used is usually limited by the memory bandwidth. For this reason, a relatively large cache memory is often assigned to each processor.

With respect to loosely coupled systems, there are three main architectural models that have emerged to date: the workstation/server model, the processor pool model, and the integrated model.

1.3.1 Workstation/Server Model

The majority of Distributed Systems currently in use are based on this model. Each user is provided with a workstation, where the application programs that he wants to use are executed. The need for workstations is based primarily on user interface requirements in application tasks. In order to enable users to share information and in general files, file servers and directory servers are used. Also, in order to allow users to share expensive devices such as high quality printers, tape drivers etc. specialized device drivers are used. The workstations are integrated by the
use of communication software that enables them to access the same set of servers. Servers are in charge of handling the special cases that need to be considered when resources are shared (authentication, security, concurrency etc.).

1.3.2 Processor Pool Model

In this model programs are executed in a set of computers managed as a processor service. Users are provided with terminals instead of workstations, connected to the network via concentrators and interacting with programs via a terminal access protocol. The pool processors usually consists of a processor with enough memory to load and run any of the system or application programs available in the system. Initially, the terminals are connected to a server that manages and allocates pool processors. When a user requests an application program, a loader server loads the executable program from the file server into a pool processor the user is assigned. The requested application programs can even be conventional operating systems adapted to work in a distributed environment. When an operating system has been loaded into the user’s pool processor, the user can interact with it directly issuing commands to load and execute programs. The Cambridge Distributed Computing System is an example of this type of distributed system architectures. The processor pool model has a better utilization of resources and flexibility than the workstation/server model. However, it does not satisfy the needs of high performance interactive programs. To overcome this disadvantage designers have gone into the implementation of hybrid systems where workstations as well as pool of processors are used. In this way, workstations can be used for interactive use and pool processors can be used to run tasks, for example, that are too big for the workstations. The Amoeba system developed at Vrije University in the Netherlands is an example of this type of hybrid architecture.
1.3.3 Integrated Model

In this model multi-user computers and workstations are integrated into a single computing system. Each computer is provided with appropriate software to enable it to act as a server and as an application processor. The system software located in each computer is similar to an operating system for a centralized multi-user system with the addition of networking software. So, every computer deals with its own applications and services. When a program is to be executed the system decides on a computer in which to run it. Most usually the computer where the request was made is chosen. Next the system locates the executable program file and loads it into the selected computer. File names are global allowing the user to refer to them using the same naming scheme. The mapping from user names to identifiers used in deciding ownership is also uniform throughout the system. An example of this type of architectures is the Locus system and the Newcastle Connection.

1.3.4 Comparing Different Architectures

The underlying architecture of a distributed system can be characterized by a number of parameters that describe its performance, reliability and functionality. These parameters are related to the communication mechanisms, the type and frequency of failures, the achievable parallelism, access latency to the storage devices, etc. Among the communication-related parameters the more important are: the overhead for inter-thread communication and the communication bandwidth (the available data transmission throughput). Among the failure related parameters the most important are: the failure and repair rate, the type of processor and stable storage failures (fail stop processors etc.), and the effect of failures (whether or not processors fail independently). The available number of processors on a processing node describes available CPU parallelism in the node. The available number of nodes in the network describes the avail-
able internode parallelism. The access latency to storage devices describes the delay associated with read and writes and the feasible streaming data rates. The above parameters are used to compare the different Distributed System Architectures and their implementations. According to the application programs to execute most frequently, some parameters are more important than others.

1.4 Distributed Operating Systems

A distributed operating system has been defined [54] as one that looks like an ordinary centralized operating system, but runs on multiple independent CPU's. The key concept is transparency, the use of multiple CPU's is invisible, i.e., transparent to the user. The user views the system as a virtual uniprocessor, not as a collection of independent machines. In addition, a Distributed Operating System does not have any single point of failure. In no case the failure of a part brings down the system.

Operating systems in conventional computers provide several important services to the application programs. These are:

- file system management and file access facilities
- peripheral device handling
- user authentication and access control
- memory and processor resource allocation;
- creation and scheduling of processes

In centralized systems these tasks are usually performed by the operating system kernel. In a distributed system these system services are located in several separate computers and are
performed by separate software components. All the tasks performed by conventional operating systems are required in distributed systems and in addition each computer must also include software to support communication on a local network. Figure 1.2 shows the main layers of a distributed operating system.

Distribution in a distributed operating system is based primarily on the use of servers designed to be used by clients programs running in workstations and other client computers. The file system, for example, runs as user-level processes separate from the operating systems used in the individual computers and can communicate directly with the user-level programs. Examples of distributed systems kernels are Amoeba, Mach, and V-kernel. The common characteristic in these kernels is the use of lightweight tasks in addition to the use of normal of heavyweight tasks (the term light or heavy refers to the amount of overhead involved to switch between different tasks). Lightweight tasks are processes (threads of control) that live in the same environment of
their parent process and share their address space via shared memory. In this way interprocess communication is very easy and flexible.

The amoeba kernel was developed at the Vrije University in Amsterdam. It was designed to be a basis for an open system with many of the components of a conventional operating system, such as the file service, outside the kernel. There are several servers: a block server, a directory server, a boot server, a loader server, and a database server. All of the computers in the amoeba system run this kernel. The kernel includes facilities for creating processes and interprocess communication based on a triple of message passing primitives designed to support Remote Procedure Call messages. These primitives are: request for use by the clients to make remote calls; get_request and put_reply for the servers to receive and respond to service calls.

The Amoeba architecture consists of four principal components: the workstations, one per user, the pool processors, the specialized servers, and the gateways which are used to link Amoeba systems at different sites. Amoeba currently runs on a collection of 24 Motorola 6810 computers connected by a 10-Megabyte-per-second local network. Figure 1.3 shows the Amoeba system architecture.

The Mach kernel was developed at Carnegie Mellon University. It is an open system based on a lightweight kernel running in each computer with services such as the file system, network service and process management outside the kernel. These services replace the system calls found in traditional operating systems. The model provided by Mach is a service model in which objects are managed by servers, and clients make requests for operations on objects by using remote procedure calls. Remote procedure calls are supported by efficient and flexible interprocess communication facilities in the kernel.

The V kernel is a research project at Stanford University. It performs functions similar to the ones performed by a software back plane in the sense that provides an infrastructure
Figure 1.3: The Amoeba Architecture
for components (for hardware, boards; for software, processes) to communicate and nothing else. Consequently, most of the facilities found in traditional operating systems such as the file system, resource management and protection are provided by V servers outside the kernel. The V system consists of a collection of workstations (SUN's), each running an identical copy of the V kernel. The kernel consists of three components: the interprocess communication handler, the kernel server (for providing basic services, such as memory management), and the device server (for providing uniform access to I/O devices). Some of the workstations support an interactive user, whereas others function as file servers, print servers, and other kind of servers. Figure 1.4 illustrates a V-kernel configuration.
Chapter 2

CONCURRENT OBJECTS AND CONCURRENT SYSTEMS

A concurrent system consists of a collection of sequential processes that communicate through shared objects. This model corresponds to a shared memory multiprocessor system where processors communicate by reading from and writing to a shared memory board. Throughout this chapter this model is used. A concurrent object is a data structure shared by several processes. Every object has a type. The object’s type defines the set of possible values it can adopt and the set of primitive operations available to manipulate it. Sequential objects are usually specified through a set of axioms that define the meaning of the operations when they are invoked one at a time by the processors. Specifications for concurrent objects are more complex. More than one operation may be invoked at a time and it is necessary to define all possible interleavings of the operation invocations.
2.1 Some Definitions for Concurrent Systems

Shared registers, shared lists, shared queues are examples of concurrent objects. The following concepts will be used to discuss the specification and implementation of concurrent objects:

Operation Any type of action on an object by a processor, such as read, write enqueue, dequeue, etc. Operations consist of two events: the operation invocation, and the operation response. The operation invocation is denoted by \( x \ op(args) \ A \), where \( x \) is the name of the object, \( op \) is an operation name, \( args \) are the arguments needed by the operation, and \( A \) is the name of the processor invoking the operation. The operation response is denoted by \( x \ term(res) \ A \), where \( term \) is a terminating condition (successful or failed) and \( res \) is the result(s) of the operation. A response matches an invocation if their object and process names agree. A pending invocation is an invocation with no matching response.

History A history is a finite sequence of operation invocation and response events. It is used to model the execution of a system (concurrent or sequential). A history is sequential if each invocation is immediately followed by a matching response. A process subhistory, \( H|P \), of a history \( H \) is the sequence of events in \( H \) whose process name is \( P \).

A history \( H \) induces an irreflexive partial order \( \prec_H \) on operations. If \( e_0 \) and \( e_1 \) are two operations belonging to \( H \) then:

\[ e_0 \prec_H e_1 \text{ if response}(e_0) \text{ precedes invocation}(e_1) \text{ in time in } H. \]

\( \prec_H \) captures the "real time" precedence ordering of operations in \( H \). In this chapter \( \prec_H \) will be used to denote temporal precedence relations among operations belonging to a history \( H \). Notice that for sequential histories \( \prec_H \) is a total order. Since \( \prec_H \) is an irreflexive partial order relation the following must hold: (\( e_0, e_1, \text{ and } e_2 \) are operations belonging to a history \( H \))

1. \( e_0 \not\prec_H e_0 \)
2. if $e_0 \prec_H e_1$ then $e_1 \not\prec_H e_0$

3. if $e_0 \prec_H e_1$ and $e_1 \prec_H e_2$ then $e_0 \prec_H e_2$

Operations that are not related by $\prec_H$ are concurrent operations. Lamport [36] introduced the symbol — $\Rightarrow_H$ to relate concurrent operations in a history $H$. The symbol — $\Rightarrow_H$ means "may affect" and has the following interpretation:

- An operation $e_0$ affects the outcome of another operation $e_1$ if $e_0 \prec_H e_1$. For example, if $e_0$ is a write operation and $e_1$ is a read operation to some shared register $Reg$, $e_0$ affects $e_1$ because it makes $e_1$ return the value it wrote to $Reg$. Two or more concurrent operations to an object may or may not affect their outcomes depending on the implementation of the object. For the above example, if $e_0$ and $e_1$ are concurrent, it is not possible to tell which value will be returned by $e_0$. It may return the old register’s value, the value $e_0$ is writing to the register, or a corrupted value due to the interference of $e_0$. The concurrent operations $e_0$ and $e_1$ are denoted by $e_0 \Rightarrow_H e_1$.

Example:

Consider two processes $P_1$ and $P_2$ communicating through a shared register $Reg$. $Reg$ consists of 3 bytes $Reg_1$, $Reg_2$, and $Reg_3$. $V$ denotes all the different values that $Reg$ can adopt, and $V^i$ denotes the $i^{th}$ value. A write operation by process $P_i$ is denoted by $Reg \text{ Wr}(V^i) P_i$ and $Reg \text{ Wr.ok.}() P_i$ for operation invocation and response, respectively. Similarly, a read operation is denoted by $Reg \text{ Rd}() P_i$ and $Reg \text{ Rd.ok.}(V^j) P_i$. Consider the following history of events, $H$:

$Reg \text{ Wr}(V^1) P_2$

---

1 Lamport uses the term "system execution" instead of history
Figure 2.1: The sequence of read and write operations by the processes $P_1$ and $P_2$

\[\begin{align*}
\text{Reg \,} &\text{Wr.} \text{ok.()} P_2 \\
\text{Reg \,} &\text{Rd}(V^1) P_1 \\
\text{Reg \,} &\text{Rd.ok.()} P_1 \\
\text{Reg \,} &\text{Wr}(V^2) P_1 \\
\text{Reg \,} &\text{Rd(} P_2 \\
\text{Reg \,} &\text{Wr.} \text{ok.()} P_1 \\
\text{Reg \,} &\text{Rd.ok.(} V^3) P_2 \\
\end{align*}\]

Figure 2.1 shows the execution of the operations in time. From Figure 2.1 and the history $H$, $e_2$ and $e_3$ are concurrent.

The partial order $\prec_H$ induced by the history $H$ is

\[e_0 \prec_H e_1 \prec_H e_2, e_0 \prec_H e_1 \prec_H e_3.\]
Concurrent operations $e_2$ and $e_3$ cannot be related by $<_H$. $e_2 \rightarrow^H e_3$ and $e_3 \rightarrow^H e_2$. The value $V^3$ returned by $e_3$ can be $V^1$, $V^2$ or a corrupted value.

To see how a corrupted value can be returned, consider the following situation:

$V^1 = 8, 7, 5$ (values from left to right contained in Reg$_1$, Reg$_2$, and Reg$_3$, respectively), and $V^2 = 4, 2, 1$. Assume that during the execution of $e_2$, $P_1$ writes 4 to Reg$_1$ and then control is switched to $P_2$ which starts executing $e_3$. $P_2$ reads Reg$_1$ (whose value at that moment is 4), Reg$_2$ (whose value is still 7) and then the scheduler schedules $P_1$ again. $P_1$ writes 2 to Reg$_2$ and 1 to Reg$_3$ finishing $e_2$ and returns the control to $P_2$. $P_2$ finishes $e_3$ by reading Reg$_3$ (whose value is 1 now) and returns the values it read from Reg$_1$, Reg$_2$, and Reg$_3$. But, this value is 4, 7, 1, a corrupted value which it was never intended to be written to the register. Notice that $e_2$ and $e_3$ can be scheduled in several ways and whether $e_2$ will affect $e_3$ or not depends on the interleaving of the operations on the three registers Reg$_1$, Reg$_2$, and Reg$_3$.

Lamport [36] proposed the following set of axioms to provide the properties of the temporal precedence relations $<_H$ (-) and $\rightarrow^H$ induced by a history $H$ ($e_i$'s denote operations belonging to a history $H$):

1. The relation $\rightarrow^H$ is an irreflexive partial ordering.

2. If $e_0 \rightarrow^H e_1$ then $e_0 \rightarrow^H e_1$ but $e_1 \rightarrow^H e_0$.

3. If $e_0 \rightarrow^H e_1 \rightarrow^H e_2$ or $e_0 \rightarrow^H e_1 \rightarrow^H e_2$ then $e_0 \rightarrow^H e_2$.

4. If $e_0 \rightarrow^H e_1 \rightarrow^H e_2 \rightarrow^H e_3$ then $e_0 \rightarrow^H e_3$.

5. For any $e_i$, the set of all $e_k$ such that $e_i \not\rightarrow^H e_k$ is finite.
2.2 Specification of objects

A sequential history for an object can be summarized by the value of the object at the end of the history. To reason about the value of an object the axiomatic specifications are used. A specification is a set of axioms of the form:

\[
\{ P \} \\
\text{op(args)/term(res)} \\
\{ Q \}
\]

where \( P \) is a precondition on the value of the object and the argument values (args) to be met for the operation invocation (op), and \( Q \) is a postcondition upon return for the given termination condition (term). A sequential history \( H \) is legal if for all object subhistories \( H \mid x \) of \( H \), each operation in \( H \mid x \) satisfies its axiomatic specification (assuming a systems with several concurrent objects). Axiomatic specifications however have no meaning for histories that are not sequential. For concurrent systems the result of an operation may depend on how it is interleaved with other concurrent operations. In one situation, however, it is possible to reason about a concurrent object using axiomatic techniques for sequential objects. This happens when for every history of the concurrent object in the system it is possible to find an equivalent legal sequential history. To find the equivalent sequential history, the concurrent operations are rearranged in time according to some definite order. The order must be consistent with the values the object is observed to contain in time. This means that the implementation of the object must not permit all possible interleavings of operations but only the ones that generate histories for which it is possible to find sequential equivalents. The exact specification about the permissible interleavings determines the correctness conditions for the implementation. Different correctness conditions have been proposed for the implementation of concurrent objects, namely, sequential
consistency [35], serializability [45] and linearizability [30]. Linearizability provides the strictest conditions. It requires an object to be implemented in such a way as if every operation on the object has been executed instantaneously between its invocation and response. This implies that concurrent object operations can still be specified with pre and post conditions. Notice that linearizability requires the implementation of the object to respect the order in which no overlapping operations were invoked.

2.3 Register axioms

Misra [38] analyzed in great detail the implementation of shared (concurrent) registers, a type of concurrent object. He provided an axiomatic definition of this type of registers, reasoning in terms of the registers' hypothetical values. Misra proposed to build a register such that to an external observer concurrent operations (read and write) on the register appear to be executed sequentially and nonconcurrent operations are executed in the same order they were requested. He shows that if a register obeys a certain set of axioms then it may be analyzed as a serial device where all the operations are sequential. To illustrate his idea consider the following example:

Example:

A history $H_a$ of read and write operations performed on a one bit register $Reg$ (flip-flop) is the following:

\[
\begin{align*}
Reg W(0) & \quad P_1 \\
Reg W. ok() & \quad P_1 \\
Reg W(1) & \quad P_2 \\
Reg R() & \quad P_1 \\
Reg R. ok(1) & \quad P_1
\end{align*}
\]
Figure 2.2: The operations by the three processes on a register. It is possible to rearrange the operations in such a way (as shown in Figure 7) that the nonconcurrent operations preserve their order and the read operations return values that are consistent with the write operations.
The order of execution of the operations in time is shown in Figure 2.2. In Figure 2.2, operations are denoted by $OP_i(val)$, where $OP$ can be $Wr$ for a write or $Rd$ for a read. The subscript $i$ is the number of the process executing the operation and $val$ (0 or 1) is the argument for the operation. Notice that $Wr_2(1)$, $Rd_1(1)$, and $Wr_3(0)$ are concurrent operations. If the register behaves as in Figure 2.2 then its behavior is indistinguishable from a register where operations take place sequentially one at a time as in Figure 2.3. Notice that in the sequence of Figure 2.3 the values returned by the read operations are consistent with the write operations.

The sequential history $H_b$ corresponding to the operations in Figure 2.3 is the following:

$Reg \, Wr(0) \, P_3$
$Reg \, Wr.ok(.) \, P_3$
$Reg \, Wr.ok(.) \, P_2$
$Reg \, Rd(.) \, P_2$
$Reg \, Rd.ok.(0) \, P_2$

$Reg \, Wr(0) \, P_1$
$Reg \, Wr.ok(.) \, P_1$
$Reg \, Wr(1) \, P_2$
$Reg \, Wr.ok(.) \, P_2$
$Reg \, Rd(.) \, P_1$
$Reg \, Rd.ok(1) \, P_1$
$Reg \, Wr(0) \, P_3$
$Reg \, Wr.ok(.) \, P_3$
$Reg \, Rd(.) \, P_2$
$Reg \, Rd.ok.(0) \, P_2$
Consider now the following history $H_c$ for the same register $Reg$ (shown in Figure 2.4):

$Reg \text{ Wr}(0) \ P_1$

$Reg \text{ Wr.ok}() \ P_1$

$Reg \text{ Wr}(1) \ P_2$

$Reg \text{ Rd}() \ P_1$

$Reg \text{ Rd.ok}(1) \ P_1$

$Reg \text{ Wr}(0) \ P_3$

$Reg \text{ Wr.ok}() \ P_3$

$Reg \text{ Wr.ok}() \ P_2$

$Reg \text{ Rd}() \ P_2$

$Reg \text{ Rd.ok}(1) \ P_2$
Figure 2.4: The operations by three processes on a register. These operations cannot be rearranged in time to return consistent values because $Rd_1(1)$ cannot precede $Wr_3(0)$. 

\[ \text{Process} \]

\[ \text{Time} \]

$P_1$, $P_2$, $P_3$
The history \( H_c \) is almost the same as the history \( H_b \). The difference is that the last read operation returned a 1 instead of a 0. For history \( H_c \) it is impossible to rearrange the operations in such a way that the nonconcurrent operations preserve their order and the read operations return values consistent with the write operations (notice that \( Rd_1(1) \) has to precede \( Wr_3(0) \)).

Misra analyzed the sequences of operations that he called schedules\(^2\) and stated that a schedule is valid if its effect is equivalent to the effect of some sequential schedule where operations are executed serially. Formally he defined a valid schedule \( S \) to be one for which it is possible to find a permutation \( S' \) satisfying a set of validity conditions. These validity conditions are (modified to the notation used in this paper):

1. For every operation \( OP \) in \( S' \) the \( OP \)'s invocation precedes the \( OP \)'s response.

2. Every pair of operations in \( S' \) has to be nonconcurrent.

3. If operation \( OP_1 \) precedes operation \( OP_2 \) in \( S \) then \( OP_1 \) precedes \( OP_2 \) in \( S' \).

4. In \( S' \) every read operation has a preceding write operation, and if \( Wr_i(x) \) is the closest preceding write operation to \( Rd_j(y) \) in \( S' \) then \( y = x \).

If a set of processes accesses a register in such a way that the schedule for the operation executions is valid, it is possible to analyze the register behavior as if the operations were not concurrent. Referring to the example discussed earlier in this section and Figures 2.2 and 2.3, \( H_a \) is a valid history (schedule) with an equivalent permutation \( H_b \).

\(^2\) Schedules are equivalent to histories.
2.3.1 The Axioms

Misra [38] proposed a set of axioms that defines how the registers should behave under concurrent read and write operations so that it is possible to analyze the outcome of such operations as if they were nonconcurrent. The set of axioms requires a register to behave in the following way:

1. If a read operation $Rd_i(x)$ returns the value $x$, then at some point within the time interval it was executed in, the register value was $x$. This means it is impossible to return a value $x$ if the register never had that value.

2. The value stored in the register previous to a write operation must be different than the value stored in it after the operation (he assumes a writing operation never writes the same value twice). This means a write operation always changes the value of the register.

3. If the register value is $x$ at some point in time, there must exist a write operation $Wr_i(x)$ that writes this value in the register.

4. The register does not change its value spontaneously. That is, if the register has the value $x$ at two different points in time $t_1$ and $t_2$ then its value is also $x$ at any time between $t_1$ and $t_2$ (again assuming that the write operations always write different values to the register).

Misra proves that every schedule $S$ for which the axioms are satisfied is valid and consequently, it is possible to find the permutation $S'$. He also shows that all the above four axioms he proposes are necessary and independent from each other.
Chapter 3

ATOMIC REGISTERS

The concept of atomic registers was introduced by Lamport and is closely related to the Concurrent Reading and Writing Problem. Concurrent read and write operations to a register must be handled carefully to prevent consistency problems. If a reading process accesses a register while a writing process is modifying it, the reader may obtain a corrupted value (an inconsistent value). Also, if two or more writing processes write to the register at the same time, these concurrent operations may leave the register with a value that was never intended to be written. Mutual exclusion is the most obvious solution for this problem. This approach, however, has several drawbacks. The most important one is that it implies the existence of some waiting time between the request of access and the execution of the read or the write operation. Wait-free solutions for the problem that allow the parallel execution of the read and the write operations are implemented in several ways. The construction of atomic registers is the most refined version of these solutions. An atomic register is a construction that simulates an ideal register where concurrent reads and writes are executed in some definite order. This order of execution must insure that the values returned by the read operations are consistent with the write operations. This chapter is devoted to the description of atomic registers and the techniques for the spec-
ification of these concurrent data structures. Techniques for the verification of the correctness of an atomic register implementation are also discussed. Besides the atomic register the regular and the safe registers are also described. These are considered weaker types in the sense that they cannot handle all the cases of concurrent reading and writing.

3.1 LAMPORT’S REGISTER CLASSIFICATION

Registers were classified by Lamport [36] into safe, regular, and atomic registers, according to their behavior under concurrent operations.

3.1.1 Safe Register

In this register a read nonconcurrent with any write obtains a correct value (the most recent one). In case of concurrent reads and writes, the read operations may return any value. So, if the register has \( n \) bits, the returned value may be any value between 0 and \( 2^n -1 \). It is the weakest type of register.

3.1.2 Regular Register

A regular register is also a safe register but in case of concurrent read and write operations, the register will return either the new value being written or the old (previous) value stored in the register. In general, a read overlapping a series of write operations will return the register’s value before the beginning of the write operations or one of the values being written.

3.1.3 Atomic Register

An atomic register is also a safe register, but under concurrent read and write operations it behaves as if they had occurred in some definite sequential order, even if the operations were concurrent. Among the restrictions for its behavior, the most important one is that in case
Figure 3.1: A reader process and a writer process accessing a shared register. The type of the register determines which values are legal for a read operation to return when it overlaps in time a write operation of two or more successive read operations \(Rd_i, Rd_{i+1}, Rd_{i+2}, \ldots\) to the register that overlap a series of successive write operations \(Wr^j, Wr^{j+1}, Wr^{j+2}, \ldots\), the case of \(Rd_i\) returning the value written by \(Wr^{j+2}\) and later read operations \(Rd_{i+1}\) and \(Rd_{i+2}\) returning previous values written by \(Wr^j\) and \(Wr^{j+1}\) is forbidden.

To illustrate the different types of registers an example is given below:

**Example:**

Consider the sequence of read and write operations on an eight bit register as shown in Figure 3.1. If the register is safe then the read operation \(Rd_i(x)\) returns value \(x = 3\), and \(Rd_{i+1}(y)\) and \(Rd_{i+2}(z)\) return values between 0 and 255. For a regular register, \(Rd_i(x)\) returns value \(x = 3\), \(Rd_{i+1}\) and \(Rd_{i+2}(z)\) return either 3 or 11. For an atomic register, \(Rd_i(x)\) returns \(x = 3\), and \(Rd_{i+1}(y)\) and \(Rd_{i+2}(z)\) may return any of the following pairs of values: (3,3), (3,11), (11,11), ...
where the first value of the pairs corresponds to \( y \) and the second to \( z \). The case of \( Rd^{i+1}(y) \) returning \( y = 11 \) and \( Rd^{i+2}(z) \) returning \( z = 3 \) is forbidden for the atomic register. However, a regular register would allow this. The safe type of register is usually implemented by hardware. The regular and atomic type have to be constructed using the safe registers and designing convenient access protocols. In general, the construction of a register must fit a combination of the following categories:

1. Safe, regular or atomic.
2. Boolean or multivalued.
3. Single reader or multireader.
4. Single writer or multiwriter.

This gives twenty four different types of registers that can be built. The weakest construction is a safe, boolean, single reader, single writer register, the strongest is an atomic, multivalued, multireader, multiwriter register. Constructions of these registers are described in the following sections.

### 3.2 SAFE AND REGULAR REGISTER CONSTRUCTIONS

The constructions discussed in this section were developed by Lamport [36]. The weakest type of register (safe, boolean, single reader, single writer) is assumed to be hardware implemented. All the constructions discussed here are also assumed to be single writer.

#### 3.2.1 Multivalued Safe Register

The implementation of a multivalued \((2^n\text{ valued})\) multireader safe register \( M \) consists of a set \((m_1, \ldots, m_n)\) of boolean multireader safe registers. The writing process writes a binary value
$V_1, \ldots, V_n$ to the register by assigning to each $m_i$ the corresponding value $V_i$. Any reading process gets the register's value by reading every $m_i$ starting with $m_1$.

### 3.2.2 Multireader Boolean Regular Register

The construction of such registers consists of a multireader boolean safe register $S$ and a variable $old$ internal to the writer's program (not shared). The write operation of the boolean value $b$ (0 or 1) to the register is executed by assigning $b$ to it if this value is different from the previous one ($old$ keeps this value). Because the only possible values for register $S$ are 0 and 1, if the writer is changing its value and there is a concurrent read operation then the returning value will always be the old value or the one being written. Therefore, $S$ behaves as a regular register.

### 3.2.3 Single Reader Multivalued Regular Register

The algorithm presented in [46] could be used to simulate this register. However, it does not use weaker types of registers strictly because some of the flags have to be atomic. For this reason, Lamport's algorithm [36] is discussed here despite being less space efficient. For a register with values in the range from 1 to $n$, the construction consists of a set of $n$ boolean multireader regular registers $r_1, \ldots, r_n$. The register's value is kept using unary encoding, where a value $n$ is encoded by setting register $r_n$ to 1 and registers $r_{n-1}$ to $r_1$ to zero. The read operation reads each register starting with $r_1$ and stopping when a 1 is found. One thing to note is that the write operation is done from right to left and the read operation is done from left to right. Lamport proves that any read operation concurrent with a writing will return only the previous value or the value just being written.
3.2.4 Multireader Multivalued Safe or Regular Register

An \(m\)-reader, \(2^n\)-valued safe or regular register \(R\) is implemented using \(m\) single reader, \(2^n\)-valued registers \(r_1, \ldots, r_m\). Each \(r_i\) keeps a copy of the register’s value that the writer makes for every reader. The write operation \(R := v\) is executed by assigning to each \(r_i\) the value \(v\), starting with \(r_1\). The \(j^{th}\) reader executes a read operation by getting the value of \(r_j\). If registers \(r_1, \ldots, r_m\) are regular, the construction implements a multireader, multivalued, regular register. If the registers are safe, the construction is also safe.

3.3 ATOMIC REGISTER CONSTRUCTIONS

In the previous section the implementation of safe and regular registers was discussed. The basic strategy used in that implementation was to build the complex registers in terms of simpler ones. In this way, multivalued multireader safe and regular registers were built in terms of safe boolean single reader registers.

3.3.1 I/O Automatons

A formal way to describe the implementation of a data object in terms of simpler ones is to model the objects as I/O automata. An I/O automaton is a non-deterministic automaton represented by a tuple \((E, Q, S, Q^0)\) where:

- \(E = E^{in} U E^{out}\) is a set of input and output events.
- \(Q\) is a finite or infinite set of states.
- \(Q^0\) is a distinguished set of starting states.
- \(S\) is a transition function of the form \((q, e, q')\), where \(q, q' \in Q \cup Q^0\) and \(e \in E\). The tuple \((q, e, q')\) is called a step and means that an automaton in state \(q\) moves to state \(q'\) after the
occurrence of event \( e \). The execution of an I/O automaton is represented by a sequence of steps called a schedule.

Processes in a concurrent system can also be modeled as I/O automats. The process will then be defined as the execution of a procedure. To model the communication between processes and objects, the I/O automats are extended with the addition of ports. This extended I/O automaton is called port automaton. A port automaton [49] is formally represented as a tuple \((V, P, CH, M)\), where:

- \( V \) is the set of values that can be sent as messages.

- \( P \) is the set of ports. Each port has a type which is master or slave.

- \( CH \) is the set of channels. Each channel is a pair of ports, one of type master and the other of type slave. A port belongs to at most one channel. A port that does not belong to a channel is called an external port. The other ports are called internal.

- \( M \) is an I/O automaton. Events in this automata are of the form \((v, p)\) where \( v \in V \) and \( p \in P \).

Communication between port automats is done through I/O channels. Messages sent from master ports to slave ports are called commands and correspond to output events of master ports and input events of slave ports. Messages sent from slave ports to master ports are called responses and correspond to output events of slave ports and input events of master ports. A port automaton is said to be well formed if any schedule of events corresponding to its ports starts with a command and consists of alternating commands and responses. A schedule of events is said to be balanced if for every command in the schedule there is a response. A procedure corresponds to a port automaton with at most one slave port. An object corresponds
to a port automaton with no master ports. Objects are specified by describing their external ports and stating all their legal schedules.

A composition of several port automata is done by defining new channels that model the interaction among the different components. An object $OBJ$ composed of other objects $OBJ_1, OBJ_2, OBJ_3, ..., OBJ_n$ is a port automaton built by interconnecting a number of port automata such that every automaton is a procedure, or corresponds to one of the objects $OBJ_1, OBJ_2, OBJ_3, ..., OBJ_n$. Formally, atomic objects are defined as follows [30]

- An object is atomic if for every schedule of events $H$, corresponding to any of its external ports there exists schedules $H'$ and $S$ such that $H'$ is a balanced extension of $H$, $S$ is a sequential schedule consisting of the same events as $H'$, and $<_{H'} \subseteq <_S$.

### 3.3.2 Atomic Operations

The notion of atomicity is strongly related to the idea of a total order in a history of operations in a system. When operations are assumed to be atomic, the execution of an operation $op_1$ can affect the execution of another operation $op_2$ only if $op_1$ precedes $op_2$. When a data object is implemented in terms of smaller objects, the operations on the object are implemented in terms of the operations on its components. These complex data objects can be described at two levels: at the high level the description is done through axioms that define the set of operations available to manipulate the object, and at the low level the implementation of the high level operations in terms of the operations executed on the components (the low level operations) is described. In Section 2.1 an example of two processes communicating through a shared register was given. The shared register $Reg$ consists of three smaller registers $Reg_1, Reg_2$, and $Reg_3$.

---

1. This definition has been modified to fit the notation used in this document.
2. $S$ is called linearization of $H$.
3. The word history is used when referring to the execution of a system and the word schedule when referring to the execution of an automata.
each one with a size of one byte. When a process wants to read from or write to Reg, it has to individually access each of Reg’s components and read from or write to it. Therefore, a read (write) operation to Reg consists of three primitive read (write) operations to Reg’s components.

Atomicity requires that when operation executions are viewed at the high level, they appear to be executed sequentially even if they are concurrent. Since the high level operations must be indivisible, the corresponding low level operations cannot be interleaved in time. Using the same example discussed in this section, consider a write operation Wr to Reg executed by process P1 and a read operation Rd to Reg executed by a process P2. As explained before each operation consists of three primitive operations to Reg’s components. These low level operations are denoted by Wr1, Wr2, Wr3 for write operation Wr and, Rd1, Rd2, Rd3 for read operation Rd (the subscript indicates the component which is being accessed). If operations Rd and Wr are atomic, the case shown in Figure 3.2(a) cannot happen. In Figure 3.2, the execution of the low level operations corresponding to Rd and Wr are interleaved in time, i.e., Rd and Wr are not executed atomically. Figure 3.2(b) shows two ways these operations can be executed atomically. The interleaving shown in Figure 3.2(a) cannot be allowed because the read operations may return inconsistent values. So, the operations must be executed strictly sequentially.

Simulation of Atomic Reads and Writes

Serial execution is, however, a very strong requirement. Fortunately, as discussed in Section 2, it is possible to implement the object such that it always behaves as if there were a serial execution of the operations even though sometimes the operations are executed concurrently. For concurrent read and write operations the object must behave as if they had occurred in some definite order because this is the way the true atomic read and write operations would have been executed. Non-concurrent operations must be executed by the system in the same order as they are requested to maintain consistency with the order observed externally. For example, if it was
Figure 3.2: (a) Concurrent read and write operations executed non-atomically. (b), (c) Two different ways to execute the operations of part (a) atomically.
observed that a write operation $W_{r_1}$ preceded in time a write operation $W_{r_2}$, the object in its attempt to simulate the serialization of operation executions cannot behave as if $W_{r_2}$ preceded $W_{r_1}$.

Consider the concurrent operations in Figure 3.3(a). The system cannot behave as if the operations would have been executed in the order $Rd_2, W_{r_1}, Rd_1$ because it contradicts the actual order of request of $Rd_1$ and $Rd_2 – R_1$ precedes the execution of $R_2$. The correct way to simulate the serial execution is to make the object behave as if the concurrent operations had been executed at some point in the time interval between its invocation and the response. For the concurrent operations in Figure 3.3(a), to simulate the serial execution of the operations, the object may behave in different ways as shown in Figure 3.3(b)(c)(d) (The asterisk denotes the point in time where the operations are assumed to take place).

It is easy to check if the object is pretending serialization correctly by looking at the values returned by the read operations. Because the operations are simulating atomic ones, a read operation must return the value written by the latest preceding write operation in the simulated order. For example, in the third situation in Figure 3.3 (part (d)), $Rd_1$ must return the value written by $W_{r_1}$ and so does $Rd_2$.

**Conditions for Simulating Atomic Read and Write Operations**

It should be obvious at this point that there cannot be physical concurrent execution of atomic read and write operations. However, this can be simulated by simulating serialization of execution of the operations. The object will behave such that to an external observer all the operations are executed sequentially according to the following observations [21]:

1. Non-concurrent read and write operations will be executed in the object in the same order (in time) as they are requested.
Figure 3.3: (a) Two consecutive read operations to a shared register overlapping in time with a write operation. (b) (c) (d) Three different ways of simulating serial execution of the operations in part (a).
2. Concurrent read and write operations are executed in some definite order such that the object behaves as if the operations had been executed at some point in the time interval between their invocation and response.

3. Reading operations must return the value written by the latest preceding write according to the simulated order.

3.3.3 Implementing Atomic Registers

As discussed in previous sections two operations are defined on an atomic register: atomic read and atomic write operations. These atomic operations are implemented by lower level primitive operations. The history of a system execution consists of a set of primitive operation executions (also called events). If a total order can be established among the events of a system execution, the system execution is defined to be atomic. The main interest is, however, to see if it is atomic from the high level point of view. From an abstract point of view, the general requirements for atomicity are:

A1. Every high level operation execution must consist of a finite nonempty set of primitive operation executions.

A2. Each primitive operation execution must belong to one and only one high level operation execution.

A3. For any primitive operation execution, there must be a finite number of other operations that precede it in time.

One operation execution precedes another if all the primitive operation executions corresponding to the first one precede all the primitive operation executions of the second. From this precedence relationship, it can be determined if the total order can be extended to the high level
operation executions. If this is possible then the system execution is also atomic from the high level point of view.

From the implementation point of view an atomic register consists of a set of real registers along with some programs that access them. Any process that needs to read from or write to the register invokes these programs. The concurrency of read and write operations occurs when two or more of these programs are invoked at the same time. In this situation the atomic register must behave as if the programs had been called in some definite order. A system execution then consists of a series of calls to the reading and writing procedures. The events in the system execution correspond to the events or primitive operation executions that implement the procedure calls. The main restriction for the read and write procedures is that they must not have loops or waiting statements because this could prevent processes from accessing the register for an indefinite time. This is implied by the requirement A1 (every high level operation execution must consist of a finite nonempty set of primitive operation executions). Also there has to be an initial write operation that sets the register with its initial value before any read operation. This follows from the requirement A3 (in any system execution, for any primitive operation execution there must be a finite number of other primitive operation execution that precede it).

An implementation of an atomic register is correct if for every system execution, a total order can be obtained in the corresponding history of read and write operations to the register. This total order must be consistent with the partial order defined by non-concurrent operations in the history. In addition every read operation must return the value written by the latest preceding write operation.
Establishing an Order among Operation Executions

To make a total order among the operations in a history of a system execution, a function is defined to map the operations onto the set of integer numbers. It is then possible to order them in time by assuming that given any two operations, the one with the smaller number is executed first. If the order obtained is total, the system execution is atomic and the construction is correct.

In a history of a system execution if there is only one writer, all the write operations are totally ordered in time. Each execution of a writing operation can be assigned a version number that represents the order in which it occurs in time. If in a system execution the write operation is executed k times then the write operations will be assigned numbers from 1 to k \( (W_{r_1}, W_{r_2}, \ldots, W_{r_k}) \). In this way it is possible to distinguish between different executions of the write operation. The value 0 is assigned to the write operation that writes the initial value the register had at the beginning of the history. This write operation \( W_{r_0} \) is assumed to have preceded all read operations.

A function \( G \) is defined to assign these version numbers to the read operations according to the following general rule:

- A read operation is assigned the version number of the write operation that wrote the value the read returns.

A construction is proved to be atomic if for every system execution the function \( G \) satisfies the following requirements (note that a system execution consist of a set of read and write operations):

G1. For every read operation \( Rd \) there is a write operation \( Wr \) whose version number is assigned as \( G(Rd) \).
G2. The write operation $W_{r_{G(Rd)}}$ precedes the read $Rd$, but the write operation $W_{r_{G(Rd)+1}}$ does not precede $Rd$.

G3. For any two read operations $Rd$ and $Rd'$, if $Rd$ precedes $Rd'$ then $G(Rd) \leq G(Rd')$.

The existence of function $G$ satisfying the above requirements ensures that the read operations return values that were written to the register, and it is possible to obtain a total order among the read and write operations of the system execution. Also, the construction will behave as if the operations had been executed in that order with every read returning the value written by $W_{r_{G(Rd)}}$. The requirement G3 is very important in obtaining an order for the operations. Consider a scenario where a read operation $Rd$ gets the value written by $W_5$ and a later read operation $Rd'$ gets the value written by $W_4$. In this case $G(Rd)$ would be 5 and $G(Rd')$ would be 4 and this would imply that $W_5$ preceded $W_4$ which is a contradiction. So, the existence of a mapping function ensures that a construction is atomic. A formal proof is given in [36].

Reading Errors

Peterson and Burns [13] defined a similar function called indexed value function $V$ that assigns a number to each operation in a system execution. The write operations are assigned the version numbers, so that no two write operations being assigned the same number. A read operation is assigned the version number of the write operation that wrote the value it returned, with a condition that the write operation must have preceded the reading. By using this function they proposed the following theorem:

- A construction is correct if for all of its system executions, there is an indexed value function $V$ such that there cannot be four (not necessarily distinct) high level operations $G$, $G'$, $H$ and $H'$ with $V(G) = V(G')$ and $V(H) = V(H')$, where $G$ precedes $H$ and $H'$ precedes $G'$.
The proof of this theorem can be found in [13]. A knowledge graph (directed graph) is built where the nodes are named with unique labels for every read and write operation. For example, if two values read or written were assigned labels \( A \) and \( B \), then the graph contains two nodes named \( A \) and \( B \). The edges in the graph represent the precedence relations between the values. If there is an edge from \( A \) to \( B \) then there exists \( op(A) \) and \( op(B) \) (\( op \) is a write or a read operation) such that \( op(A) \) preceded \( op(B) \). It is shown that a cycle in the graph implies nonatomicity, and the graph is acyclic if and only if the system execution is atomic. Informally, if the graph is acyclic then it is possible to obtain a total order among the operations which implies that the system execution is atomic. Since an operation can be either a writing or a reading, it is necessary to analyze sixteen different cases. However, simplifications can be made, and for the case of a single writer the sixteen cases can be collapsed into two which are:

1. \( W_I \) preceding \( W_J \) and \( R_J \) preceding \( R_I \), with \( V(W_I) = V(R_I) \) and \( V(W_J) = V(R_J) \).

2. \( W_I \) preceding \( W_J \) and \( W_J \) preceding \( R_I \), with \( V(W_I) = V(R_I) \).

These two bad cases are considered reading errors and have specific names. The first one is called the \textit{new-then-old} reading error. It consist of a read operation returning a new (most recent) value from the register and a strictly later read operation returning an old (previous to the most recent) value. The requirement G3 for the function \( G \) insures that this error does not happen. The second one is called the \textit{out-of-date} reading error. It consist of a read operation returning an old value even though the most recent write operation that had placed a new value in the register was completely finished before the reading. Condition G2 for the function \( G \) prevents this error from happening. The second type (\textit{out-of-date}) of reading error is very easy to prevent in atomic constructions with a single writer because every write operation completely erases the previous value.
Chapter 4

Sharing Memory in Asynchronous Message Passing Systems

In this chapter we present an algorithm to simulate Read-Modify-Write Registers in a message passing system with unreliable asynchronous processors and asynchronous communication. The algorithm works correctly in presence of a strong adversary that can stop up to \( T \) processors or stop the delivery of their messages where \( T = \lfloor N/2 \rfloor - 1 \) and \( N \) is the number of processors in the system. This is the best resilience that can be achieved in the message passing systems. The high resilience of the algorithm is obtained by using randomized consensus algorithms and a robust communication primitive. The use of this primitive allows a processor to exchange local information with a majority of processors in a consistent way and therefore, take decisions safely. The simulator makes it possible to translate algorithms for the shared memory model to that for the message passing model. With some minor modifications the algorithm can be used to robustly simulate shared queues, shared stacks, etc.
4.1 Introduction

In the shared memory model for intercommunication in multiprocessor systems, several synchronization primitives have been proposed to coordinate actions among the processors. The most common use of these primitives is to resolve conflicts when there are concurrent requests of access to a shared resource. These primitives are also used when the processors need to select some unique values, e.g., id's, time stamps, etc. On the other hand, the agreement problems in which all processors select the same value can also be solved using these synchronization primitives. In all cases the purpose of a primitive is to let a processor execute some specific operation without being interrupted. The type of operations we are referring to always involves access to a shared data object and the uninterrupted (atomic) access to it is absolutely necessary to guarantee the correct execution of the operation.

The most basic synchronization primitives are the atomic read and write instructions. These instructions guarantee that concurrent reads and writes to the object, will be executed in a serial manner according to some definite order to maintain consistency of the shared data. A thorough study of these primitives can be found in [12], [15], [13], [44], [46], [28], [52], [53].

A very useful primitive is the read-modify-write instruction which atomically reads an object and writes a new value that is a function of the current value. Instructions that belong to this category are Test-and-Set, Fetch-and-Add, swap, and, compare and swap. A good description of these primitives can be found in [27], [33], [28].

Another category of primitives includes memory to memory operations. Examples of this type are the move instruction which atomically copies the value of one shared object into another, and the multiple assignment instruction which assigns values to more than one shared data objects atomically. Synchronization can also be obtained through the use of shared queues, shared stacks, shared lists etc. [28].
There are several ways to implement these primitives in shared memory architectures. In a system with a large number of processors, the shared memory board is connected to the processors through a multistage interconnected network. To resolve conflicts among concurrent operations to the same memory location a technique called combining is used. With this technique, two messages requesting to execute operations \( f \) and \( g \) to the same shared variable arriving at a network switch are combined into one. The operation the new message carries is a combination of \( f \) and \( g \) (\( f \circ g \)). When the new combined message arrives to the memory block, the result of the combined operation is stored in the variable. Next, the old content \( (v) \) of the variable is returned in a reply message to the switch. The switch remembers which message arrived first, and returns \( v \) to the corresponding sending processor. The other processor is returned \( f(v) \) (or \( g(v) \)). The combining technique is very useful to implement several types of read-modify-write instructions and was used in the construction of the New York University Ultracomputer [27].

Can we use these concepts to achieve synchronization in message passing systems? Bar-Noy and Dolev [4] gave the first step in this direction by proposing building blocks — one for the shared memory model and one for the message passing model. If an algorithm for the shared memory model is written using the building block, it is possible to translate the algorithm to work in the message passing model by using the equivalent block, and vice versa. One of their goals is to identify the basic elementary operations in the two models and develop a general translation scheme between them. In a more recent work [3] the authors have presented an algorithm to emulate atomic single writer multireader registers in a message passing system.

One important implication of this translation scheme is to prove that if there is a solution to a given problem in one of the models then a solution to the same problem can be developed in the other model too.
4.2 Model

A message passing multiprocessor system can be characterized by a set of parameters that describes the behavior of its components. These parameters model the environment in which an algorithm will work. The type of parameters of special importance in this paper is the one that describes the degree of synchronism in the system among the processors and in the communication mechanism. The algorithm presented in Section 4 handles asynchronous processors and asynchronous communication. The message delivery order however, must be synchronous. This could be relaxed but it would complicate the description of the communication primitives. In addition, we assume that the processors have fail-stop type of failures.

Processors in a system are said to be asynchronous if any processor can wait an arbitrary amount of time between two of its own steps. If processors are synchronous then there is a constant \( \phi \) such that in any time interval in which some processor makes \( \phi + 1 \) steps all nonfaulty processors make at least one step.

The communication in the system is asynchronous if messages can take an arbitrary amount of time to be delivered. Synchronous communication implies that there is a constant \( t_d \) such that every message sent has to be delivered within time \( t_d \) to its destination. Another type of asynchrony in the communication is the message delivery order. If messages can be delivered out of order then the message delivery order is said to be asynchronous. If messages are delivered in the same order as they were sent then the delivery is synchronous. Notice that the synchronous message delivery order guarantees that if two messages are sent by two different processors to the same destination, the one that was sent at the earlier time arrives first.

Processor and communication asynchrony are potential sources of nondeterminism that an algorithm must be able to handle when solving a specific problem. Another source of nondeterminism is the type of processor failures that the system can have. Two types of processor
behavior under failures are usually considered: Fail-stop and Byzantine. A processor has fail-stop behavior if any failure makes the processor stop doing any computation or sending messages. A processor has Byzantine behavior if a failure makes it behave erratically, making wrong computations or sending contradicting or corrupted messages.Obviously, the first type of processors is easier to deal with than the second one.

All these sources of nondeterminism in the system are well represented by a game between an adversary and a given algorithm. The goal of the adversary is to make the algorithm fail. For that purpose the adversary is assumed to have some or all the following privileges:

- Instruct the processors when to operate or even to stop up to $T$ processors and not to let them restart at all (processor asynchrony). The factor $T$ is a resilience parameter. If an algorithm can stand up to $T$ processor failures and continue producing correct results then it is $T$-resilient.

- If a processor is waiting for $N-1$ messages from other processors, the adversary can suspend up to $T$ messages (communication asynchrony), $T = \lfloor N/2 \rfloor - 1$, where $N$ is the total number of processors.

- Dictate the delivery time for messages sent and thus make them arrive out of order (message delivery order asynchrony).

- Arrive at the decisions by observing the messages to be sent and the internal state of all the processors. If however, an algorithm takes random steps like flipping a coin, the adversary should not be able to predict the outcome of future random steps.

In the next section, a communication primitive that allows processors to exchange local information robustly is discussed.
4.3 Communication Primitive Exchange

The processors use the primitive Exchange to exchange their knowledge. Exchange will ensure that the global information collected by a processor is consistent with that by a majority of processors. Exchange requires every processor to have an array to store information about all processors in the system including itself. By invoking Exchange, a processor sends its array and receives the arrays from other processors. This allows the processor to update its local information. In order to know which information is the most recent one the timestamp must be used. The timestamps are generated locally by the processors. With all the elements in the array timestamped, it is possible to compare them and keep the most recent one.

Notation

For this point onward, \( Ts(Array A[i]) \) will denote the timestamp of the element \( i \) in the array \( A \). The procedure Exchange for any processor is given below:

Procedure Exchange (Myarray)

begin

\[ \text{Counter} := 0; \]

\[ \text{Done} := false; \]

while not Done do begin

Broadcast(Myarray);

Exit := false;

while not Exit do begin

Receive(Otherarray);

\{ Otherarray is sent by a processor \( P_j, j \epsilon [1..N] \} \}

if Less(Otherarray, Myarray) then

Discard(Otherarray);
else

    if Equal(Otherarray, Myarray) then begin

        Counter := Counter + 1;

        if Counter >= N - T then Exit := true;

        { N is the number of processors, T \leq N/2 - 1 }

        end-if

    else begin

        Update(Myarray, Otherarray);

        Exit := true;

        end-else;

    end-while;

    if Counter >= N - T then Done := true;

    Return();

end-Exchange;

Procedure Update (Array1, Array2)

begin

    For j:=1 to N do

        if Ts(Array1[j]) < Ts(Array2[j]) then Array1[j] := Array2[j];

end-Update;

Procedure Less (Array1, Array2)

begin

    Isless := true;

end-Less;
while Isless and $j \leq N$ do

    if $Ts(Array1[j]) > Ts(Array2[j])$ then $Isless := false$;

end-Less;

Procedure Equal (Array1, Array2)

begin

    Isequal := true;

    while Isequal and $j \leq N$ do

        if $Ts(Array1[j]) \neq Ts(Array2[j])$ then $Isequal := false$

    end-Equal;

Note that the indices of the arrays correspond to the processors. Therefore, the same index
$(j)$ of two arrays are used to compare them. Procedure Exchange assumes that a majority of
the processors will always remain connected with each other. A processor that gets completely
disconnected from the majority of processors is isolated by them. When the processors invoke
Exchange, they all will have an empty slot in their arrays corresponding to the disconnected
processor. Therefore, they will agree in that there is no information available for that processor
and will continue with their jobs without taking the disconnected processor in consideration.

The following lemmas show that a processor belonging to a majority will succesfully execute
Exchange in a finite time:

Lemma 3.1

Every invocation of the procedure Exchange terminates.

Proof
By assumption, the adversary can stop up to $T$ processors from sending messages or hide up to $T$ messages that were sent by good processors. Every processor is then guaranteed to receive at least $N - T$ messages containing arrays from other processors. This will allow the processor to update its array with the most recent information. This implies that at some point of time a majority of processors will have the same information about others. After every update a processor sends its array and collects $N - T$ arrays from other processors. When the majority of processors finish updating their arrays, these $N - T$ arrays will be identical. At this point the procedure terminates.

End of proof

The following lemma shows that the arrays a processor obtains by using Exchange are linearly ordered in time. This implies that decisions taken based on the information in the array are safe because they are done according to the most recent information about the system.

Lemma 3.2

Let $X$ and $Y$ be two arrays collected by processor $P_i$ at times $t_0$ and $t_1$, $t_0 < t_1$. Then for all $j \in [1..N]$, $X[j] < Y[j]$.

Proof

Assume that $X$ and $Y$ were collected in two consecutive invocations of the procedure Exchange by $P_i$, and $X[j] \geq Y[j], j \in [1..N]$. This implies $\exists k \mid X[k] > Y[k]$.

Processor $P_i$ passed $X$ as the parameter to Exchange which returned $Y$. In the procedure Exchange, $X$ can only be changed by calling the procedure Update. The procedure changes the value of element $X[k]$ to $Y[k]$ only if it finds an element $Z[k]$ of one of the collected arrays such that $Z[k] > X[k]$. This contradicts the assumption that $Y[k] < X[k], k \in [1..N]$.
4.4 Simulator

The algorithm in this section simulates Read-Modify-Write registers. This type of registers allows the atomic execution of the following procedure: \( \text{Reg} \) is a Read-Modify-Write register)

**Procedure RMW**

begin

\[ \text{Temp} := \text{Reg}; \]

\[ \text{Reg} := f(\text{Temp}); \]

return(\text{Temp});

end-RMW;

In the above procedure RMW, \( f \) is a function from register values to register values. If \( f \) is the increment function then the register is a fetch-and-add register. If \( f \) sets the content of the \( \text{Reg} \) to 1 only when \( \text{Temp} \) is equal to 0 then register is a test-and-set register. Other type of Read-Write-Registers include Swap, Compare-and-Swap, etc. To simulate a Read-Modify-Write register in a message passing system every processor keeps a copy of the register. Since several processors may attempt to execute a RMW operation at the same time, it must be ensured that all processors execute the same operation on their local copies. So, at any time, all the copies have the same value and the consistency is maintained. In our algorithm, when a processor wants to execute an operation, it broadcasts a request message including its \( ID \). The message also contains a timestamp to distinguish old request messages from new ones. Each processor maintains two set
of flags $Opreq_1$ and $Opreq_2$, $j \in [1..N]$ to keep track of the requesting processors. The request messages are collected by a procedure $Receive$. The procedure $Receive$ after receiving a message from a processor $P_k$ reads $Opreq_1k$ and writes $Opreq_2k$ with the opposite value. $Receive$ also stores the timestamp included in the message in a variable $Tstamp$. When $receive$ gets a request for the first time, it retransmits it to ensure that all processors receive it. $Receive$ knows if the request is old by comparing the timestamps. If an incoming message carries a timestamp older than or equal to the timestamp of the last message received, it is a duplicate request and must be discarded. The reason for the retransmission is that the adversary is allowed to hide messages and one of these hidden messages could be a request message. Since it is only necessary to recognize a duplicate of a request message from a given processor, the timestamps can be generated locally. Every processor has a variable called $Opnum$. This variable is incremented every time a processor executes an operation in its local copy of the register. $Opnum$ is used as the timestamp for the request messages. The procedure $Receive$ is described below ($Tstamp_j$ is the variable where $Receive$ stores the timestamp of the last request message received from processor $P_j$):

$$\begin{align*}
\text{Procedure } Receive; \\
\text{begin} \\
\text{upon receiving operation request message } <\text{ Req, Opnum, ID }> \\
\text{if } (Opreq_{1ID} = Opreq_{2ID}) \text{ and } (Tstamp_{ID} < Opnum) \text{ then begin} \\
Opreq_{2ID} := \neg Opreq_{1ID}; \\
Tstamp_{ID} := Opnum; \\
\text{Broadcast}(<\text{ Req, ID }>); \\
\text{end;} \end{align*}$$

end-receive;
To resolve conflicts when two or more processors want to execute a RMW operation at the same time, the processors are forced to reach an agreement. In the algorithm, every time a process wants to execute an operation it goes into a leader election process. If a processor is elected as the leader, it knows that all other processors agree that it is its turn to execute an operation. The leader election procedure works based on the processor ID’s. The processors reach an agreement on a single ID by going through several rounds of consensus. In order to defeat an adversary that can stop processors and hide messages, a randomized consensus algorithm must be used. It has been proven [26], [24] that the deterministic consensus algorithms cannot work correctly in the presence of even one faulty processor in completely asynchronous environments. In the Leader election procedure described in this section, a highly resilient consensus protocol is used. It was proposed in [4]. This protocol was proven to have resilience $T = \lceil N/2 \rceil - 1$ in an environment of asynchronous processors and asynchronous communication.

In the procedure Leader election a processor starts with a suggested ID. To decide on which processor will be elected, it goes into $\lceil \log N \rceil$ rounds of consensus. At every round it reaches an agreement with the rest of the processors on one bit of the leader’s ID. If a processor loses a round of consensus, it selects at random another ID among the processors that have requested operations. The ID it selects must still have the chance to be selected as the leader’s ID. The ID’s of the processors that have requested operations are kept in a set called Reqset. After selecting a new ID a processor continues with the next round of consensus. This speeds up the leader election process and makes the algorithm very resilient. The reason for the high resilience is that all processors cooperate with each other to elect the leader even after losing a round of consensus. With all the processors participating in the election, they will always be a majority that elects a leader. One interesting thing to note here is that if a processor dies after making a
request, it can still win the election. The operation it requested is carried out by the remaining good processors.

The procedure \textit{Bitconsensus} used inside \textit{Leader-election} works in the following way: A processor calls the procedure with an initial value. This value is one of the bits of the ID it is suggesting as the leader's ID. The processor maintains an array $Bcr$ with $N$ elements. Every element of $Bcr$ has three fields. The first one is \textit{Sugval} which contains the value other processor is suggesting. The second variable \textit{Round} keeps the number of attempts a processor has made to reach an agreement. \textit{Round} is initialized to zero. The third field is \textit{Timestamp} where a processor writes $Opnum$ and $Bitnum$, the index of the leader's ID bit on which the processors are trying to reach an agreement. This information will be used by the procedure \textit{Exchange} to generate labels that identify the messages. Processors exchange values and check if all are suggesting the same value. If this is true, \textit{Bitconsensus} terminates. If there is a disagreement, they flip a coin and adopt a new value according to the result. They exchange the values and flip the coin again if the disagreement persists. The processors keep repeating these steps until all flip the coin with identical values. A processor increments \textit{Round} every time it initiates a new attempt to reach an agreement. The processors are asynchronous and run at different speeds. So, some processors attempt to reach an agreement faster than others. In \textit{Bitconsensus} if a group of processors that are ahead in their number of attempts agree in their suggested value then they don't flip the coin and go into the next round with the same value. Processors that are behind in their number of attempts also adopt this value. After exchanging values, the leaders (the processors that are ahead) check if the processors that disagree in their suggested values are two or more rounds of attempts behind. If this is true, the leaders decide on their suggested values only and exit the procedure. The processors that are behind will eventually notice that the leaders have reached consensus, accept that value, and exit the procedure.
is very important to identify which processors are the leaders, because they are the ones who decide the final value. The leading processors are the ones whose Round is the largest. To determine the leaders at any given, time a processor needs to know the values of the Round variables of all other processors. The procedure Exchange does this.

Ideally if all processors flip the same coin, they all get the same result. This corresponds to flipping a globally shared coin. With such a coin Bitconsensus terminates at most in two rounds. It has been proven however [29] that a perfectly unbiased global coin cannot be built. The procedure Flip_Global_Coin implements a biased globally shared coin. For $\gamma > 1$, a biased global coin has the property that with probability greater than $\frac{\gamma + 1}{2\gamma}$ all the processors flip the same value. In the Flip_Global_Coin procedure every processor flips a local coin and depending on the result it increments or decrements a counter. The processor then gets the values of the counters of other processors and adds them up. If the result is greater than $\gamma N$ then it decides one. If the result is less than $-\gamma N$ then it decides zero. Otherwise it repeats the steps above. To keep information about other processors, every processor maintains an array Coin of size $N$. Every element in Coin has two fields. The first field Contribution keeps the value of a processor's counter. The second field is Timestamp where a processor stores the value of Opnum, Bitnum, Coinnum, and Iteration. The variable Coinnum contains the number of times a processor has flipped a coin in a given consensus round. Iteration keeps the number of times a processor has flipped a local coin while trying to decide a value in Flip_Global_Coin. These values are used by Exchange as labels to distinguish old messages from new ones. The expected number of rounds the processors will go through before flipping the same coin is $(\gamma + 1)^2 N^2$. The proofs of these results can be found in [4]. IDpool is a set containing the ID's of all processors.

MYID is the ID of the executing processor.
Procedure Leader_election(Opnum, ID);

begin
    for i:=1 to \[\log N\] do begin

        \{ ID_i and Leader_i are the \textit{i}_{th} bit of ID and Leader, respectively\}
        
        Leader_i := Bitconsensus(Opnum, i, ID_i);

        if \(\text{Leader}_i \neq \text{ID}_i\) then
            \[ID := \text{RandomSel}\{\text{NewID}\_\text{Reqset} \mid \text{NewID}_j = \text{Leader}_j \forall j \in (1..i)\}\]

        if ID = NULL then begin

            \(\text{Found} := \text{false}\);

            repeat
                for all \(\text{NewID} \subseteq \text{IDpool} - \text{Reqset}, \text{NewID} \neq \text{MYID}\) do begin

                    Requestarrived := (Opreq_2 \neq Opreq_1);

                    if Requestarrived and (\(\text{NewID}_j = \text{Leader}_j\) \forall j \in (1..i)) then begin

                        \(ID := \text{NewID};\)

                        \(\text{Found} := \text{true};\)

                        Exit for_all loop;

                    end-if;

                end-for_all;

            until found;

            end-if;

        end-for;

    end-Leader_election;
Procedure Bitconsensus(Opnum, Bitnum, Val);

begin

for j:=1 to N do begin

Bcr[j].Round := 0;
Bcr[j].Sugval := 0;
end-for;

while true do begin

Bcr[MYID].Round := Bcr[MYID].Round + 1;
Bcr[MYID].Sugval := Val;
Bcr[MYID].Timestamp :=(Opnum, Bitnum);
Exchange(Bcr);

Maxround := max<sub>1≤k≤N</sub> (Bcr[k].Round);

if ( (Bcr[MYID].round = Maxround) and (\forall k \in \{1..N\} and k ≠ MYID | (Bcr[k].Sugval ≠ Bcr[MYID].Sugval) )

( Bcr[k].Round < Bcr[MYID].Round - 1 ) ) then

return( Bcr[MYID].Sugval);
else

if (\exists v | \forall k, Bcr[k].Round = Maxround, Bcr[k].Sugval = v) then

Val := v;
else

Val := Flip_Global_Coin(Opnum, Bitnum, Bcr[MYID].Round);

end-while;

end-bitconsensus;
procedure Flip_Global_Coin(Opnum, Bitnum, Coinnum);

begin

    for j:=1 to N do begin

        Coin[j].Contribution := 0;
        Iteration := 0;

        while true do begin

            Iteration := Iteration + 1;
            Coin[MYID].Contribution := Coin[MYID].Contribution + Localcoinflip

            { Localcoinflip returns +1 or -1 with equal probability }

            Coin[MYID].Timestamp := (Opnum, Bitnum, Coinnum, Iteration);

            Exchange(Coin);

            Globalvalue := \sum_{j=1}^{N} Coin[j].Contribution;

            if Globalvalue > \gamma N then return(1);
            if Globalvalue < -\gamma N then return(0);

        end-while;

    end-FlipGlobalCoin;

The procedure used to execute operations on a processor's local copy is Execop. As explained before, at any time a processor \( P_i \) may request to execute an operation. If it succeeds in being elected the leader, the other processors execute the operation it requested in their local copies. Execop continuously checks the flags \( Opreq1_j \) and \( Opreq2_j \) to find out if any processor \( P_j \) wants to execute an operation. If these flags have different values, the procedure Receive received a request message from \( P_j \). All the requesting processors ID's are written in the set \( Reqset \) by Execop. If Execop receives a request from its local processor, it broadcasts a request message.
Then it calls *Leader.election*. Next it executes the operation requested by the winner of the election in the local copy. *Execop* also resets the request flags *Opreq* corresponding to the winning processor by reading *Opreq2* and writing *Opreq1* with the same value. If the executing processor *Pi* is not the winner, *execop* calls *Leader.election* again, and repeats the above steps until *Pi* is elected the leader. After *Pi* executes its local operation, *Execop* looks into the processors in *Reqset* and tries to elect them as leaders. By making *Execop* help other processors before executing another operation requested by its own processor, it is guaranteed that any processor succeeds in executing its operation in finite time. The local processor *Pi* running *Execop* requests to execute an operation by calling the procedure *Myrequest*. *Myrequest* signals *Execop* by reading *Opreq1* and writing *Opreq1* with the opposite value. *Myrequest* then waits until *Execop* executes the operation by winning an election round. *Execop* puts the old value of the register in a variable called *Retval*. Next, it signals *Myrequest* by writing *Opreq1* with the value of *Opreq2*. *Myrequest* gets the value from *Retval* and returns it to *Pi*. In the procedure below, the function *Randomsel* selects at random an *ID* from the set *Reqset*, *f* is a function the processors use to change the value of the register:

**Procedure Execop;**

```plaintext
var Opnum: integer;

   Reqset: Set;

   Object, Retval: Register;

   ID: integer;

begin

   Opnum := 0;

   Reqset := ∅

   while true do begin
```
for \( j := 1 \) to \( N \) do

\[
\text{if } \left( (\text{Op}_{req1,j} \neq \text{Op}_{req2,j}) \text{ and } (j \neq MYID) \right) \text{ then}
\]

\[
\text{Reqset} := \text{Reqset} + [j];
\]

\[
\text{Myop} := \text{false};
\]

\[
\text{Otherop} := \text{false};
\]

if \( \text{Op}_{req1MYID} \neq \text{Op}_{req2MYID} \) then

\[
\text{Myop} := \text{true};
\]

\[
ID := MYID;
\]

\[
\text{Broadcast(< Req, ID >)};
\]

end-if

else

\[
\text{if Reqset} \neq \emptyset \text{ then}
\]

\[
\text{Randomsel}\{\text{ID} \in \text{Reqset}\}
\]

\[
\text{Otherop} := \text{true};
\]

end-if;

if \text{Myop} \text{ or Otherop} \text{ then begin}

\[
\text{Leader} := \text{Leader.election(Opnum, ID)};
\]

\[
\text{if Leader} \neq ID \text{ then}
\]

\[
\text{wait until Op}_{req1\text{Leader}} \neq Op_{req2\text{Leader}};
\]

\[
\text{Retval} := \text{Object};
\]

\[
\text{Object} := f(\text{Retval});
\]

\[
\text{Opnum} := \text{Opnum} + 1;
\]

\[
\text{Op}_{req1\text{Leader}} := \text{Op}_{req2\text{Leader}};
\]

if \text{Leader} = MYID \text{ then}
repeat

RandomSel\{ID \in Reqset\}

Leader := Leader.election(Opnum, ID);

Retval := Object;

Object := f(Retval);

Opnum := Opnum + 1;

Opreq1.Leader := Opreq2.Leader;

if (ID = Leader) then

Reqset := Reqset \{ID\};

until Reqset = \emptyset

end-if;

end-while

dend-execop;

Procedure Myrequest;

begin

Opreq2.MYID := \neg Opreq1.MYID;

wait until Opreq2.MYID = Opreq1.MYID;

return(Retval);

end-myrequest;
4.5 Correctness

To prove the correctness of the simulation algorithm of the Read-Modify-Write registers we need to prove that it satisfies the following conditions:

C1) Any processor will execute a requested operation in finite time

C2) A requested operation is not executed more than once.

C3) If a processor executes an operation in its local copy, the other processors execute the same operation in their local copies. This guarantees that the processors will see the same value for the object at any time and the consistency is maintained

Lemma 4.1

Every invocation of the procedure Leader.election terminates.

Proof

A process executing Leader.election starts with an initial ID passed to it by Execop. The processor goes through \([\log N]\) rounds of consensus to get its initial ID selected as the leader’s ID. If it loses any consensus round, it selects another ID from Reqset that can still be selected. If all processors in Reqset have lost the election, it waits to receive the request message(s) from the processor(s) that won the last consensus round. This message is guaranteed to arrive in finite time because every processor retransmits request messages received from others. When the message arrives, the processor adopts the ID of the sender, and continues executing Leader.election. All processors participate in every consensus round until the last bit of the leader’s identifier is selected. If some processors fail, by our assumption a majority of \(N - T\) processors will always be working to select a leader. Therefore, the procedure Exchange will always terminate. This implies that Bitconsensus and consequently, Leader.election will also terminate.
End of proof

Lemma 4.2

Any processor will succeed to be a leader within a finite number of rounds of leader election.

Proof

A processor $P_j$ enters a leader election round after broadcasting a request message. If none of the $N - 1$ remaining processors is attempting to be a leader, when they receive $P_j$’s request they will enter the process of leader election with $P_j$’s ID and will elect $P_j$ the leader. If one or more of the remaining processors want to be the leader, they will include $P_j$’s ID in the set $Reqset$ after receiving the request from $P_j$. When all processors receive $P_j$’s request, $P_j$’s ID will be in $Reqset$ of all processors. As explained earlier, after a processor succeeds in executing an operation it requested, it helps the processors in $Reqset$ to be elected the leaders. Because $P_j$’s ID is in every $Reqset$ set, eventually it will be elected the leader. It is not possible to know the exact number of rounds of leader election that will be executed before $P_j$ is elected the leader since $P_j$’s request messages may take an arbitrary amount of time to arrive to their destination. If the messages arrive instantaneously, $P_j$ would succeed after at most $2N - 1$ rounds of leader election.

End of proof

Lemma 4.3

Every invocation of the procedure $Myrequest$ terminates.

Proof

When $Myrequest$ is invoked, it signals $Execop$ by writing $Opreq2$ the opposite value of $Opreq1$. It then waits until their values become equal. $Execop$ will set them equal after electing its
processor a leader. Lemmas 4.1 and 4.2 guarantee that any processor becomes the leader in a finite time. Then $\text{Execop}$ sets $Oreq_1$ and $Oreq_2$ equal. This allows $Myrequest$ to terminate.

End of proof

**Theorem 4.1**

Condition $C_1$ is satisfied.

**Proof**

A processor requests to execute an operation by calling $Myrequest$. Since $Myrequest$ terminates in finite time (Lemma 4.3), the processor also executes its operation in finite time.

End of proof

**Theorem 4.2**

Condition $C_2$ is satisfied.

**Proof**

An operation requested by a processor $P_j$ is executed more than once only if another processor $P_i$ receives a duplicate of $P_j$'s request and takes it as a new request. $P_i$ then enters the leader election process with $P_j$'s ID and elects it the leader. But, a duplicate message will not be accepted. The request messages are timestamped. An incoming message is recognized as a new request of $P_j$ only if it carries a timestamp greater than that of the last request message received from $P_j$.

End of proof

**Lemma 4.3**

For every invocation of $\text{Leaderelection}$ at most one leader is elected.
Proof

The processors ID's are assumed to be unique and therefore, differ in one or more bits. In every round of Leader election the ID's that do not correspond with the bit selected are discarded. The processors suggesting those ID's select new ones among the ones that can still be selected as the leader's ID. Since all ID's are different, as processors go through the rounds of consensus, the number of ID's that can be selected become fewer and fewer. In the last round they will have at most two choices. One of these ID's loses the election and the other becomes the elected leader's ID.

End of proof

Theorem 4.3

Condition C3 is satisfied.

Proof

Every time a processor wants to execute an operation, it goes through the process of leader election. Every invocation of Leader election elects at most one processor (Lemma 4.3). In Execop all processors execute the operation requested by the elected leader (processor) only. Since there is only one leader, all processors execute the same operation in their local copies.

End of proof

Theorem 4.4

The algorithm simulates Read-Modify-Write Registers in a message passing system

Proof

Theorems 4.1, 4.2 and 4.3 show that conditions C1, C2 and C3 are satisfied. This proves the correctness of the algorithm.
Theorem 4.5

The algorithm works correctly in presence of up to \([N/2]-1\) processor failures.

Proof

It is necessary to prove that the good processors will be able to execute their operations when up to \([N/2]-1\) processors have failed. To execute an operation a processor needs to be elected the leader. It first broadcasts a request message and then calls the procedure \textit{Leader\_election}. Good processors will be able to execute operations if the procedure \textit{Leader\_election} always terminates. After a leader is elected, all good processors execute the operation requested by the leader.

In order to reach an agreement on every bit of the leader’s ID, \textit{Leader\_election} calls the procedure \textit{Bitconsensus}. To decide a value \textit{Bitconsensus} needs to collect information about the other processors. For this purpose it calls the procedure \textit{Exchange}. To collect information about all processors \textit{Exchange} needs to communicate with a majority of processors. Information about a processor will be stored in at least one node belonging to this majority. If only up to \([N/2]-1\) processors fail, there is always a majority of processors that can exchange information. This collected information is used by \textit{Bitconsensus}. The latter eventually decides a value and terminates. The procedure \textit{Leader\_election} that calls \textit{Bitconsensus} \([\log N]\) times also eventually terminates and a leader is elected. The good processors therefore will be able to execute their operations.

End of proof
4.5.1 An example

Consider a message passing distributed system with 4 processors: \( P_1, P_2, P_3, \) and \( P_4 \). These processors need to generate unique labels \(^1\) such that when a processor selects one, no other processor can select the same. One solution to this problem is to have a Fetch-and-add register. Every time a processor needs a label, it executes a fetch-and-add(1) instruction to the register and uses the returned value as the label.

In the simulation, every processor keeps a local copy of the fetch-and-add register. These registers are: \( R_1, R_2, R_3, \) and \( R_4 \) belonging to processors \( P_1, P_2, P_3, \) and \( P_4 \) respectively. The request flags are initialized as \( Opreq_{1j} = Opreq_{2j} \) for the processor \( P_j, j = 1 \) to \( N \). \( Opnum \) for every processor is initialized to zero and \( Reqset \) is initialized as an empty set.

Assume that \( P_1 \) and \( P_4 \) want to execute a Fetch-and-add(1) operation at the same time. Both call \textit{Myrequest} which signals \textit{Execop} by setting \( Opreq1 = \neg Opreq2 \). \textit{Execop} sends request messages to all other processors because this operation is requested by its own processor. \( P_1 \) sends request messages to \( P_2, P_3, \) and \( P_4 \). \( P_4 \) sends request messages to \( P_1, P_2, \) and \( P_3 \). \textit{Execop} running in processors \( P_1 \) and \( P_4 \) ignore the request messages because they give priority to the requests made by their own processors. \textit{Execop} puts the I.D. of any requesting processor in \( Reqset \). The procedure \textit{Receive} in processors \( P_2 \) and \( P_3 \) gets the request message and sets \( Opreq_{11} \neq Opreq_{21} \) and \( Opreq_{14} \neq Opreq_{24} \). \textit{Execop} running in \( P_2 \) and \( P_3 \) selects at random one of the requests and calls \textit{Leader.election} with the ID of the selected processor. Assume that \( P_2 \) selects the request made by \( P_1 \) and \( P_3 \) selects the request made by \( P_4 \). At this point all processors are in the process of electing a leader. The processors will go into two rounds of consensus to select the two bits of the leader's ID. Because the processors try to reach an agreement taking random steps (flipping a coin) it is not possible to know before hand which

\(^1\)These labels can be timestamps used to resolve the order of access of several processors to a data base
one will be elected the leader. Suppose $P_1$ is elected the leader. Every processor executes the operation requested by $P_1$ in their local copies. Also, they reset the request flags belonging to $P_1$ by writing $Opreq_1 = Opreq_{21}$. $Execop$ in processor $P_1$ has the ID of $P_4$ in its $Reqset$. So, $P_1$ tries to help $P_4$ to get elected the leader. Other processors have $P_4$'s request also. So, they call $leader\_election$ with $P_4$'s ID. Because all processes call the $Leader\_election$ procedure with $P_4$'s ID, $P_4$ is elected the leader. All processors execute the operation requested by $P_4$. 
Chapter 5

Distributed Simulation using ISIS

In this chapter we discuss the use of the ISIS Distributed Programming Toolkit for the simulation of distributed algorithms. We start with a brief description of the ISIS system and then we describe the way we have used it to simulate the algorithm presented in Chapter 4. ISIS is a UNIX-based programming environment designed to provide the programmer with a set of tools that makes simple and easy to write adaptive and reconfigurable solutions to applications that must stand the occurrence of failures in the processing units or in the communication mechanisms. Its nice features become evident when writing solutions to a problem that require coordinated and cooperative work from a set of processors, and when we need to manage replicated data.

5.1 The ISIS System

ISIS is based on processes and messages. An ISIS process can be defined as an address space containing one or more lightweight tasks. Processes communicate through messages. Different type of messages are handled by separate tasks. Task execution is FIFO and non-preemptive but they can wait on, and signal condition variables when required. ISIS stands certain type of
failures like losing messages but cannot handle network partitions, it also supports reconfigura-
tion and continued execution after crash failures but as long as the crashing sites or programs
just stop executing and do not send corrupted messages or go into infinite loops. ISIS supports
a virtually synchronous environment, in which processes can be structured into process groups.
Events such as broadcasts to the group, changes in the group membership, and state transfer ap-
pear to occur (to the members of the group) synchronously, or in other words instantaneously.
This is of particular importance if we consider that there are cases in which the result of a
distributed computation is affected by the order in which events are observed. When several
processes are cooperatively working in the solution of a problem, if broadcasts to the processors
arrive in different order, or if failures are observed in different order by different processors in
the group, inconsistencies may arise. By supporting a virtually synchronous environment, ISIS
offers the possibility of writing programs as if they were going to work in an ideal setting. The
compiler is in charge to produce code that is able to run in a realistic environment. ISIS runs
programs designed for synchronous environments relaxing any kind of synchronization that the
algorithm does not rely upon. This helps to execute programs in a more parallel fashion. The
implementation of the idea of virtual synchrony is based on a careful analysis of the ordering
requirement of the application being run.

The programming tools offered by ISIS reflect the intention of the designers in addressing
a particular set of applications in Distributed Systems. The type of application addressed is
the one that in order to be implemented efficiently needs to be decomposed into orthogonal
components that can be treated separately. ISIS stresses in solving the problems caused by the
asynchronous propagation of information among processes. Processes learn about the occurrence
of a particular event through the arrival or not arrival of messages. This implies that processes
in a system may observe events to occur in different order if the messages arrive in different
order. This is a potential source for inconsistency problems because there are cases in which a distributed computation is affected by the way in which events are observed. The tools have orthogonal functionality. They permit the programmer to break up an application into components that can be solved independently and extended gradually into a complete system.

According to its use an ISIS tool belongs to one of the following categories:

- **Tools for managing process groups.** They include tools for creating, joining, or leaving group processes. Also, it includes mechanisms for obtaining group views, for monitoring group changes, for atomically transferring the state of the group to a new member, etc.

- **Tools for process and process group communication.** ISIS defines a new type of object called a *message*. A message is manipulated like an input/output stream and is usually sent to an entry point defined by another process. For communication with process groups ISIS provides a set of broadcast and reply mechanisms.

- **Tools for managing replicated data.** ISIS provides mechanisms to maintain replicated data using a broadcast to update but reading locally. Processes do not need to block when doing a read update or releasing a lock.

- **Synchronization tools.** ISIS provides these tools for cases in which processes executing concurrently need locking and mutual exclusion mechanisms to avoid interference between concurrent activities.

- **Tools for organizing a distributed computation.** There are three ways in which we can organize a distributed computation using ISIS. A first scheme, called *coordinator-cohort* scheme, selects some member to be responsible for a request. Non-coordinator processes function as passive backups, taking no action unless a failure occurs. The second scheme, called *redundant*, is one where all members of a group execute a request in parallel, pre-
sumably arriving at identical results and changing the replicated data in identical ways.
The third scheme, called subdivided, consists of having each member perform one part of
the request, with the collected outcome being presented to the caller.

- Tools for detecting failures and recovering after they occur. Mechanisms are provided for
detecting failures and informing any interested party of a failure. To allow failed processes
to recover, mechanisms are provided for creating periodic checkpoints or logs that can
be replayed on recovery. The recovery mechanisms take in consideration the possibility
of total or partial failures. In case of a total failure, if all processes that make up an
application fail, it is possible to restart the whole application gracefully using its stable
storage. In case of a partial failure, if one of the processes executing an application fails, it
is possible to reintegrate the failed process into the system and transfer the current system
state to it.

- Other tools. ISIS also provides tools for supporting transactions and protection.

All the tools are fault tolerant and support virtual synchrony. In case of a process failing at
the time it is performing some action using the tools, either the action is executed completely
or not executed at all. For example, if a process fails while executing a broadcast to a group,
then all the processes receive the message or none of them. The tools support virtual synchrony
in the sense that by using the tools all processes in the system observe the same events and in
the same order. This applies not only to message delivery events, but also to failure recoveries,
group membership changes, etc.

As discussed before, the main problem to solve if we want to ensure that processes observe
events in the same order is, to ensure that messages arrive in the same order. For this reason, ISIS
provides a set of four multicast primitives. The difference between them is the order in which the
broadcasted messages are delivered to their destinations with respect to other messages sent to
the same destinations. The stronger the message delivery order accomplished by the multicast
primitive, the more expensive in terms of overhead it is. The multicast primitives provided are
the following:

**fbcast** It is the least costly broadcast primitive; it provides FIFO ordering on a point to point
basis and is reliable in the sense that either all the destinations or none of them receive it
even if the sender fails.

**cbcast** It provides a generalized kind of FIFO ordering. All messages sent using this primitive
are delivered in the order in which they were sent. This means that if a process $P_1$
broadcasts messages $M_a$, $M_b$ and $M_c$ then all the destinations will receive first message
$M_a$ then $M_b$ and last $M_c$. It is also guaranteed that no destination will get message $M_c$
unless all of them have previously received messages $M_a$ and $M_b$. Another thing that
ISIS guarantees is that if any invocations of cbcast are causally related, the corresponding
messages are delivered everywhere in the order of invocation. Two multicasts events are
causally related if information about the first could have reached the point where the second
was begun before it was initiated there. The primitive cbcast however does not ensure any
particular delivery order for the case of two or more independent processes broadcasting
messages using this primitive. This means that if processes $P_1$ and $P_2$ broadcast messages
$M_a$ and $M_b$ using cbcast then some destinations could receive message $M_a$ first and message
$M_b$ next and other destinations could receive the messages in the reverse order.

**abcast** This primitive guarantees a stronger delivery order that the cbcast primitive. It solves
the problem of two independent processes broadcasting messages to the same destination.
If the processes use the abcast primitive, all the destinations receive the broadcasted
messages in the same order. For example, if processes $P_1$ and $P_2$ broadcast messages
$M_a$ and $M_b$, respectively then all the destinations receive message $M_a$ first and then $M_b$.,
or in the reverse order. The order accomplished by \textit{abcast} is a consistent one but not predetermined.

\textbf{gbcast} If some processes broadcast messages using the primitive \textit{ebcast} and other processes use the primitive \textit{abcast} (to the same destinations) then again no particular delivery order is guaranteed. The \textit{gbcast} broadcast primitive solves this problem. All processes that receive a message transmitted using \textit{gbcast} observe the same order between that \textit{gbcast} and other messages they receive, regardless of the mechanism used to send those messages.

The multicast primitives are the fundamental tools that ISIS provides, other tools are built using them. In the simulation of the algorithm presented in Chapter 4 we have basically used the broadcast primitives and the tools for process group management. So we will not describe in detail the other tools that ISIS provides. As we explain how we conducted the simulation of the algorithm we will discuss the characteristics of the tools used.

\section*{5.2 The Simulation Program}

In order to test the correctness and fault tolerance of the algorithm and evaluate its performance in terms of messages exchanged and execution time, we wrote a simulation program where we implemented the procedures \textit{EXCHANGE, FLIP\_GLOBAL\_COIN}, and \textit{BITCONSENSUS}. As we explain before the running time of the algorithm is mostly determined by the consensus protocol that we use and in particular by the procedure that implements the shared global coin. Because we can only assure that with probability greater than \(\frac{2^{γ+1}}{2^γ}\) all processes will see the same outcome after they flip the global coin, in the worst case the consensus protocol will run for an infinite time. As we discussed earlier, for randomized algorithms like our consensus protocol we can calculate only the expected running time. The expected running time of our algorithm is \((γ + 1)^2 * n^2\). Both procedures \textit{FLIP\_GLOBAL\_COIN} and \textit{BITCONSENSUS} are claimed
to be fault tolerant in the sense that any process running them will be able to finish their execution even if up to \([N/2 - 1]\) processes in the system fail. Also, any process running the algorithm must obtain correct results in presence of failures in the communication mechanisms. Specifically, a process running the algorithm can expect that in any broadcast it makes to the other processors, only \([N/2 + 1]\) will arrive to their destinations.

5.2.1 The structure of an ISIS process

A process in a ISIS application is internally structured into a number of tasks. An ISIS task looks like a C function and shares the same address space and global variables as the other tasks and functions in the process. An ISIS program like a C program has a function called `main` which is the first one to be executed when we run the program. A task is implemented as a C function but has the particularity that can be invoked by the system in response to certain events, the most common of which is message delivery. A task is declared in an ISIS program with the command `isis.task(task_name, function_name)`. A task that is started up in response to a message delivery is called an entry. A process can have many entries and each one is given a different entry number. When a message is sent, it is addressed to a particular entry. On delivery, a pointer to the message is passed as a parameter to the entry which in most cases reads the content of the message and passes it to other tasks or functions. One particular thing worth mentioning about ISIS tasks is that when a task makes certain ISIS system calls (like a broadcast that waits for replies), it is possible for a new task to start and begin executing before the system call returns. The original task will later continue from where it left off. This type of system calls that allow other tasks to be started before they return are called “blocking”.

In an ISIS application, the function `main` usually just reads the command line arguments, if there is any, initializes ISIS with the system call `isis.init(portnumber)` (for our system the port
number is 1623), and sets off the main loop with the system call `isis_mainloop(task_name)`.

The argument to `isis_mainloop` is the name of the first task to be run. The general structure of the function `main` in an ISIS program is then:

```c
main()
{
  variable declaration and function prototyping

  isis_init(<port_no>);
  /* task declaration */
  isis_task(task_name, function_name);
  .
  .
  /* entries declaration */
  isis_entry(entry_number, task_name, function_name);
  .
  isis_mainloop(task_name);
}
```

When the task started by `isis_mainloop` begins executing, ISIS inhibits the delivery of messages from other processes. This ensures that you can do all the necessary initialization before the program has to respond to events. If this task remains in an infinite loop it is necessary to make the system call `isis_start_done` inside the task. This call tells the ISIS system that the start up sequence is completed. ISIS automatically invokes `isis_start_done` at the end of the
main task. But if this task remains in a infinite loop the call has to be made explicitly. Otherwise, the ISIS program remains also in an infinite loop executing only the main loop. Another system call used in the simulation program is the \texttt{isis.accept.events} call. This call explicitly tells ISIS to accept events like incoming messages. This call is a blocking call.

5.2.2 Implementation of the procedure \textit{EXCHANGE}

\textit{EXCHANGE} needs to broadcast the table passed by the procedure \texttt{FLIP.GLOBAL.COIN} and the procedure \texttt{BITCONSENSUS} and then collect the tables sent by the other processors in order to compare them with the table it previously broadcasted. After making the comparison \textit{EXCHANGE} decides if it needs to update the table or if increment a counter that keeps track of the number of tables identical to the one broadcasted that \textit{EXCHANGE} has collected. As it was discussed in Chapter 4, the basic idea is to make the processes exchange information. In this way if a message get lost the processes that received the message will pass the information to the ones that did not receive. For this reason any processor instead of just broadcasting a message containing information about itself, it broadcasts a table containing all the information collected so far about other processes in the system. The tables contain $N$ slots. Each one corresponds to one process in the system. In these slots a process stores the information collected from the other processes. Processes using \textit{EXCHANGE} can be sure that when the procedure returns, the information obtained from the other processes is representative of their current state. Remember that every time that \textit{EXCHANGE} updates its table, it rebroadcasts it, and that \textit{EXCHANGE} returns only after it has received $\lfloor N/2 - 1 \rfloor$ tables identical to the one it broadcasted.

Because we are working in an asynchronous environment, messages can arrive at any time. Processes are assumed to execute computations at different speeds and broadcast messages in
an asynchronous fashion without waiting for acknowledgments from the processes that receive the message. This implies that in general a process will receive messages when it is not ready to use it because it is executing some part of the algorithm that involve only local computations. In order to handle this problem we implemented a procedure that buffers the received messages so that EXCHANGE can access them when it is ready to use them. This procedure builds a linked list prepending each newly arrived message to the front of the list. EXCHANGE will search this list and extract the messages it is looking for. EXCHANGE search for messages by comparing timestamps. Every element in the table passed to EXCHANGE has a timestamp which is generated by the local process. EXCHANGE searches for messages in the list by comparing timestamps. Messages in the list whose timestamp matches the one passed to EXCHANGE are extracted from the list to be evaluated. Messages in the list however, will be stored in an unordered way with respect to their timestamps. To decrease the time EXCHANGE spends searching for messages in the list, in our implementation we store messages in two different lists. In one list we store the messages containing tables created by the BITCONSENSUS procedure, and in the second list we store messages containing tables created by the FLIP.GLOBAL.COIN procedure. We also use two procedures "EXCHANGE" one is used by BITCONSENSUS and the other is used by FLIP.GLOBAL.COIN. This improved the organization of our code and made it easier to debug.

The procedures that receive messages in the program are declared as entries. As explained above we have two entries. The only work this procedures does is to call a procedure that prepends the content of received message ( the table ) to the front of the corresponding list. The procedure that extracts messages from the list (called cm.receive and conse.receive ) for EXCHANGE work, as described above, by comparing timestamps. As these procedures search the list, they compare timestamp, if the timestamp of the message is older, it deletes the message
from the list. If the timestamp is greater, it leaves the message in the queue. If the timestamp is equal, it removes the message from the queue and passes it to EXCHANGE. One important thing worth mentioning about extracting messages from the list is that in the case of the messages containing tables created by the procedure BITCONSENSUS, it is necessary to check in addition to the timestamp the value of the variable round. In the procedure BITCONSENSUS every time a process go into a new round it calls EXCHANGE and consequently broadcasts its table. A slow process will have all these tables buffered, but the ones that it must consider are the ones that have the highest value for the variable round. This implies that when conse.receive is searching for messages in the list and finds a table sent by a particular process, it must search the entire list and make sure that there is no other message for which the variable round has a higher value.

5.2.3 Implementation of BITCONSENSUS and FLIPGLOBALCOIN

The implementation of these procedures is straightforward and follows exactly the steps described in the algorithms. FLIPGLOBALCOIN is called by BITCONSENSUS and the latter is called by a driver program that makes the process join a group (all processes join the same group) and initializes the variables. The complete code for the program is included as an appendix of this document.

5.2.4 Results

By running the program we verified the fault tolerant properties of our algorithm as well as it correctness. Processes were killed while they were executing the BITCONSENSUS and FLIPGLOBALCOIN procedures and this did not disrupt or prevent the remaining operational processes from finishing executing the algorithm. In no case the failure of a process was observed to provoke other processes to arrive to contradicting decisions. One interesting thing
we observed is that when the processes flipped the shared global coin, they always agreed in the coin's value. This is due to the fact that any process knows the result of the local coin flip of all others at every iteration of the algorithm. So, they must come up with the same result for the coin flip at the end of the algorithm. This is not true in a shared memory environment for which the algorithm was originally written. Due to time restrictions we are not able to test the algorithm in different circumstances and for this reason we are not providing tables with results of the simulation.
Chapter 6

SUMMARY

The main objectives of our work has been to analyze the implementation of concurrent data objects in the shared memory model, and to develop techniques for the simulations of these objects in a message passing system. The motivation for the development of simulation techniques is to translate algorithms written for shared memory systems to work in message passing systems, and to investigate to what extent and under what conditions, problems that are solvable for the shared memory model can be solved in the message passing model by simulating shared memory synchronization primitives.

The most important topics that we have addressed are:

1. The description of the shared memory model and message passing model for Distributed Systems, their advantages, and disadvantages.

2. The concepts of concurrent systems and concurrent objects. The notion of history of a system execution and the associated partial and total order relations.

3. The axiomatic techniques for the specification of concurrent objects.

4. The methods developed for the verification of the correctness of a concurrent data object
implementation.

5. The notion of atomicity and its close relation to the obtation of a total order in a history of operations in a system.

6. The concept of atomic regular and safe registers and the techniques used in their implementation.

7. The notion of I/O and Port automatons and their use to model register constructions.

8. The translation of algorithms for the shared memory model to the message passing model and the development of an algorithm for the simulation of shared memory synchronization primitives in a message passing system.

Considerable effort was devoted to explain the concept of atomic register, its implementation in the shared memory model, and to describe how a system can pretend to execute atomic read and write operations even though the execution of these operations is interleaved in time.

Safe registers are considered to be physically realizable. The regular and the atomic registers are built using them. For these type of registers, the readers and the writers execute special protocols to maintain the regularity and the atomicity conditions. From an abstract point of view, a regular or an atomic register can be visualized as an I/O automaton composed by the register and the procedure I/O automatons. The register automatons model the safe registers, and the procedure automatons model the protocols executed by the readers and the writers.

The correct simulation of atomic reads and writes requires that every operation behaves as if it had occurred instantaneously at some point in time between its invocation and its response. In addition, in a history of operation of a system it must be possible to replace the real operations with the fictitious instantaneous ones such that a read operation always returns the value written by the latest preceding write.
The implementation of an atomic register is difficult. In some cases the algorithms were incorrect and had to be modified. Intuitive proofs in these cases failed to uncover situations in which the atomicity conditions were violated.

Atomic registers are used as synchronization primitives in the solution of different problems in distributed systems. Their use can be found in the solutions to the atomic snapshot problem [1], the concurrent timestamping problem [25], etc. On the other hand, the atomic registers have their limitations. They cannot be used to implement other common synchronization primitives like test-and-set and Read-Modify-Write registers. Herlihy [28] proved this by showing that the atomic registers cannot solve the consensus problem even for 2 processes. Any synchronization primitive that can solve the 2 process consensus problem like the test-and-set and fetch-and-add register, or the general n process case cannot be implemented in terms of atomic registers. To build them other approaches have been taken [28] [49].

The algorithm proposed in chapter 4 was designed to simulate Read-Modify-Write registers in message passing systems. Algorithms for simulating other types of synchronization primitives like atomic multiwriter registers, shared queues, etc. can be obtained by making some minor modifications to the algorithm presented here. For these primitives it is necessary that a processor that wants to execute an operation passes to others the value it wants to enqueue or write. This value could be included in the message the processor broadcasts requesting to execute an operation. The receiving processor buffers this value and uses it later when the sender succeeds in being elected the leader. The algorithm proposed is very resilient. It guarantees that any processor will be able to execute an operation even in presence of up to \( \lceil N/2 \rceil - 1 \) processor failures. This resilience is achieved through the use of a robust communication primitive and a randomized consensus algorithm. Basically, the algorithm is based on the fact that if a majority of processors can always communicate and exchange information then they would be able to
decide which processors have failed. The price paid for achieving high resilience is the large number of messages exchanged and the large size of the messages. The existence of algorithms for simulation of synchronization primitives in message passing systems implies that it is possible to construct emulators that translate algorithms for the shared memory model to that for the message passing model. Also, it implies that the existence of the solution for a problem in one model guarantees the existence of a solution in the other model. After the translation is done, optimization should be done in order to reduce the amount of message exchanges. One point to emphasize here is that when translating algorithms from the shared memory model to the message passing model, the degree of resilience cannot be maintained. Chor and Moscovici [18] have proven that the shared memory model is more powerful in this respect. One direct application of this algorithm is in the implementation of resilient mutual exclusion algorithms.
Bibliography


[38] Misra, J. Axioms for memory access in asynchronous hardware systems. ACM Transactions on Programming Languages and Systems 8, 1 (January 1986), 142–153.


APPENDIX A

The Program

#include <isis.h>

#ifndef _STDIO_H
#define _STDIO_H
#include <stdio.h>
#endif

#ifndef _ERRNO_H
#define _ERRNO_H
#include <errno.h>
#endif

#include <malloc.h>
#include <string.h>

#define GET_CM 1
#define GET_CONSEMESS 2

#ifndef PORTNO
#define PORTNO 1623
#endif
#define GAMMA 3
#define NUMPRO 4
#define LOCALCOINFLIP(x) ((int)(random() % 2 == 0 ? -1 : 1))

#ifndef TRUE
#define TRUE 1
#endif
#ifndef FALSE
#define FALSE 0
#endif

/*****************

/* Type declaration for all the data structures used by the procedure FLIP_GLOBAL_COIN */

struct coin_time_stamp {
    int Opnum;
    int Bitnum;
    int Coinnum;
    int Iteration;
};

struct Mycoin {
    int Contribution;
};
int id;

struct coin_timestamp timestamp;

struct Coin_array{
    struct Mycoin pcoin[NUMPR0];
};

struct Coin_messages{
    struct Coin_array Coin_table;
    struct Coin_messages *next;
    struct Coin_messages *previous;
};

struct Coin_messages *cm_frontqueue = NULL;
struct Coin_messages *cm_endqueue = NULL;
int My_id = 0;
int Numfail;
address *gaddr_p;

/* Type declaration for all the data structures
   used by the procedure BITCONSENSUS */
struct consensus_timestamp {
    int Opnum;
    int Bitnum;
};

struct Myconsensus{
    int round;
    int sugval;
    struct consensus_timestamp timestamp;
};

struct Consensus_array {
    int id;
    struct Myconsensus BCR[NUMPRO];
};

struct Consensus_messages{
    struct Consensus_array Consensus_table;
    struct Consensus_messages *next;
    struct Consensus_messages *previous;
};

struct Consensus_messages *conse_frontqueue=NULL;
struct Consensus_messages *conse_endqueue=NULL;

/**
 * Declare tasks and entry points */

main()
{
    void group_change();
    void get_cm();
    void get_conssemess();
    void test_loop();
    void convertmsg();

    srandom(time()); /* seed random number generator */

    Numfail = ((double)(NUMPRO) / 2.0 != NUMPRO / 2 ?
               (NUMPRO + 1) / 2 - 1 : NUMPRO / 2 - 1);

    isis_init(PORTNO);
    /* Declare tasks and entry points */
    isis_task(test_loop, "test_loop");

    /* Declare the task that monitors group changes */
isis_task(group_change, "group_change");

    /* Declare entries */
isis_entry(GET_CM, get_cm, "get_cm");
isis_entry(GET_CONSEMESS, get_consemess, "get_consemess");

    /* Declare message types */
isis_define_type(10, 'r', sizeof(struct Coin_array), convertmsg);
isis_define_type(11, 't', sizeof(struct Consensus_array), convertmsg);

    /* start the ISIS main loop */
isis_mainloop(test_loop, NULLARG);
}

    /* ................................................................. */

    /* This procedure buffers messages originated by the
     FLIP_GLOBAL_COIN procedure */

int queue_cm(coin_mess)
struct Coin_array *coin_mess;
{
struct Coin_messages *newmess;
newmess = (struct Coin_messages *) malloc(sizeof(struct Coin_messages));

memcpy(&newmess->Coin_table, coin_mess, sizeof(struct Coin_array));
newmess->next = NULL;
newmess->previous = NULL;

if (cm_frontqueue == NULL) {
    cm_frontqueue = newmess;
    cm_endqueue = newmess;
}
else {
    cm_endqueue->next = newmess;
    newmess->previous = cm_endqueue;
    cm_endqueue = cm_endqueue->next;
}

/* ........................................................................................................ */

/* This procedure deletes messages originated by the FLIP_GLOBAL_COIN procedure */

delete_cm(ptr)
struct Coin_messages *ptr;
{
if (ptr->previous == NULL) {
    cm_frontqueue = cm_frontqueue->next;
    if (cm_frontqueue) cm_frontqueue->previous = NULL;
}
else {
    ptr->previous->next = ptr->next;
}
if (ptr->next == NULL) {
    cm_endqueue = cm_endqueue->previous;
    if (cm_endqueue) cm_endqueue->next = NULL;
}
else {
    ptr->next->previous = ptr->previous;
}
free(ptr);
}

/* ................................................................. */

/* This procedure extracts messages containing tables
   created by the FLIP_GLOBAL_COIN procedure */
int cm_receive(Other_coin_table, timestamp)
struct Coin_array *Other_coin_table;
struct coin_time_stamp *timestamp;
{
    struct Coin_messages *q_pointer;
    struct Coin_messages *find_first_cm();

    while ((q_pointer = find_first_cm(timestamp)) == NULL) {
        isis_accept_events(0);
    }

    memcpy(Other_coin_table, &q_pointer->Coin_table, sizeof(struct Coin_array));
    delete_cm(q_pointer);
}

/* ............................................................................ */

/* This procedure finds the message in the list that has a
timestamp equal to the one passed to it. If a timestamp
is encountered that is older than the one passed then
the message is removed from the list. Messages with
timestamps that are newer are left in the list. */
struct Coin_messages *find_first_cm(timestamp) 
struct coin_time_stamp *timestamp;
{
  int result;
  struct Coin_messages *curr;
  struct Coin_messages *old_timestamp;
  struct coin_time_stamp *temp;
  struct coin_time_stamp *max_table_timestamp(); 
  
curr = cm_frontqueue;

  while (curr != NULL) {
    temp = max_table_timestamp(&curr->Coin_table);
    result = timestampcmp(timestamp, temp);
    if (result == 0) {
      break;
    }
    else if (result == 1) {
      old_timestamp = curr;
      curr = curr->next;
      delete_cm(old_timestamp);
    }
    else {
  
  
}
curr = curr->next;

return(curr);

/*   */
/* This procedure finds the element in the table with 
the greatest timestamp. It returns that timestamp   */

struct coin_time_stamp *max_table_timestamp(coin_table)
struct Coin_array *coin_table;
{
int index;

static struct coin_time_stamp temp;

memset(&temp, 0, sizeof(struct coin_time_stamp));

for (index = 0; index < NUMPRO; index++) {
    if (timestampcmp(&coin_table->pcoin[index].timestamp, &temp) == 1) {
        memcpy(&temp, &coin_table->pcoin[index].timestamp,
sizeof(struct coin_time_stamp));
    }
}
int cm_exchange(Myarray)
struct Coin_array *Myarray;
{
    int nresp;
    int mayo;
    int Counter = 0;
    int Done = FALSE;
    int Exit = FALSE;
    static int replies[NUMPRO];
    struct Coin_array Otherarray;

    mayo=((int)(NUMPRO/2)) + 1;

    while (!Done) {
        nresp=fbcast_l("x", gaddr_p, GET_CM, "%R[1]", Myarray, mayo,
        
        "%d", replies);
        
        /* This is the procedure exchange used by
           the procedure FLIP_GLOBAL_COIN */
    
        /* This is the procedure exchange used by
           the procedure FLIP_GLOBAL_COIN */
    
        return(&temp);
    }
}
Exit = FALSE;

while (!Exit) {
    cm_receive(&Otherarray, Myarray->pcoin[My_id].timestamp);
    if (Less(&Otherarray, Myarray)) {
        /* don't do anything with msg */
    }

    else if (Equal(&Otherarray, Myarray)) {
        Counter++;
        if (Counter >= NUMPRO - Numfail) {
            Exit = TRUE;
        } /* end if */
    } /* end else if */

    else {
        Update(Myarray, &Otherarray);
        Exit = TRUE;
    } /* end else */
}

} /* end while end */

if (Counter >= NUMPRO - Numfail) {
    Done = TRUE;
}

} /* end while done */

return;
}
/* Procedures Less Equal and Update are used by 
procedure cm_exchange to compare tables an update 
the table that was passed to it */

int Less(array1, array2)
struct Coin_array *array1, *array2;
{
    int index;
    int cmpres;

    for (index = 0; index < NUMPRO; index++) {
        cmpres=timestampcmp(&array1->pcoin[index].timestamp,
                           &array2->pcoin[index].timestamp);
        if ( cmpres==1 || cmpres==0 ) return(FALSE);
    }
    return(TRUE);
}

int Equal(array1, array2)
struct Coin_array *array1, *array2;
{
    int index;

    for (index = 0; index < NUMPRO; index++) {
        if (timestampcmp(&array1->pcoin[index].timestamp,
                         &array2->pcoin[index].timestamp) != 0) {
            return(FALSE);
        }
    }
    return(TRUE);
}

Update(array1, array2)
struct Coin_array *array1, *array2;
{
    int index;

    for (index = 0; index < NUMPRO; index++) {
        if (timestampcmp(&array1->pcoin[index].timestamp,
                         &array2->pcoin[index].timestamp) == -1) {
            memcpy(&array1->pcoin[index], array2->pcoin[index],
                    sizeof(struct Mycoin));
        }
    }
}
return;
}

/* This procedure implements the algorithm for
flipping the shared global coin */

int Flip_Global_Coin(Opnum, Bitnum, Coinnum)
int Opnum, Bitnum, Coinnum;
{
  int j;
  int Iteration = 0;
  int Globalvalue;
  struct Coin_array Coin;

  for (j = 0; j < NUMPRO; j++) {
    Coin.pcoin[j].Contribution = 0;
    Coin.pcoin[j].id = 0;
    memset(&Coin.pcoin[j].timestamp, 0, sizeof(struct coin_time_stamp));
  }

  while (TRUE) {
    Iteration++;
  }
Coin.pcoin[My_id].Contribution += LOCALCOINFLIP(0);
Coin.pcoin[My_id].id = My_id;
Coin.pcoin[My_id].timestamp.Opnum = Opnum;
Coin.pcoin[My_id].timestamp.Bitnum = Bitnum;
Coin.pcoin[My_id].timestamp.Coinnum = Coinnum;
Coin.pcoin[My_id].timestamp.Iteration = Iteration;
cm_exchange(&Coin);
Globalvalue = 0;
for (j = 0; j < NUMPRO; j++) {
    Globalvalue += Coin.pcoin[j].Contribution;
}

if (Globalvalue > GAMMA * NUMPRO) {
    return(1);
}

if (Globalvalue < - GAMMA * NUMPRO) {
    return(0);
}

/* This procedure monitors group */
void group_change(gview_p, arg)
groupview *gview_p;
int arg;
{
    My_id=0;
    while(!addr_ismine(&gview_p->gv.members[My_id]))
        My_id++;
}

/* ................................................................. */

    /* This procedure is in charge of receiving 
     * the messages containing tables generated 
     * by the FLIP_GLOBAL_COIN procedure */

void get_cm(msg_p)
message *msg_p;
{
    static struct Coin_array temp;
    int index;
msg_get(msg_p, "%/R[1]", &temp);
reply(msg_p, "%d", My_id);
queue_cm(&temp);
}

/* ................................................................. */

/* Dummy function for data conversion, it is not necessary to implement one in this case because all processes are running in the same type of computers */

void convertmsg(data)
char *data;
{
}

/* ................................................................. */

/* timestampcmp works like the C function strcmp: */

0 if time1 = time2
-1 if time1 < time2
int timestampcmp(timel, time2)
struct coin_time_stamp *timel;
struct coin_time_stamp *time2;
{
    if (timel->Opnum > time2->Opnum) return(1);
    else if (timel->Opnum < time2->Opnum) return(-1);

    if (timel->Bitnum > time2->Bitnum) return(1);
    else if (timel->Bitnum < time2->Bitnum) return(-1);

    if (timel->Coinnum > time2->Coinnum) return(1);
    else if (timel->Coinnum < time2->Coinnum) return(-1);

    if (timel->Iteration > time2->Iteration) return(1);
    else if (timel->Iteration < time2->Iteration) return(-1);

    return(0);
}
/* This procedure buffers messages originated by the BITCONSENSUS procedure */

int queue_consemess(consemess)
struct Consensus_array *consemess;
{
    struct Consensus_messages *newmess;

    newmess= (struct Consensus_messages * ) malloc(sizeof(struct Consensus_messages) ) ;

    memcpy(&newmess->Consensus_table, consemess, sizeof(struct Consensus_array));

    newmess->next=NULL;
    newmess->previous=NULL;

    if (conse_frontqueue == NULL) {
        conse_frontqueue=newmess;
        conse_endqueue=newmess;
    }
    else {

conse_endqueue->next=newmess;
newmess->previous=conse_endqueue;
conse_endqueue=conse_endqueue->next;

/* ................................................................. */

/* This procedure deletes messages 
originated by the BITCONSENSUS 
procedure */
delete_consemess(ptr)
struct Consensus_messages *ptr;
{
if (ptr->previous == NULL) {
    conse_frontqueue=conse_frontqueue->next;
    if(conse_frontqueue) conse_frontqueue->previous= NULL;
}
else {
    ptr->previous->next = ptr->next;
}
if (ptr->next == NULL) {

conse_endqueue = conse_endqueue->previous;

if (conse_endqueue) conse_endqueue->next = NULL;
else {
    ptr->next->previous = ptr->previous;
}
free(ptr);

/* This procedure extracts messages containing
   tables created by the BITCONSENSUS procedure */

int conse_receive(Other_conse_table, timestamp)
struct Consensus_array *Other_conse_table;
struct Consensus_timestamp *timestamp;
{
    struct Consensus_messages *q_pointer;
    struct Consensus_messages *find_first_consemess();

    while ( (q_pointer = find_first_consemess(timestamp)) == NULL ) {
isis_accept_events(0);

memcpy(Other_conse_table, &q_pointer->Consensus_table,
       sizeof(struct Consensus.array));

delete_consemess(q_pointer);

/* This procedure finds the first message
   in the list with a timestamp that matches
   the one passed as an argument.
*/

struct Consensus_messages *find_first_consemess(timestamp)

struct consensus_timestamp *timestamp;
{

    int result;
    static struct Consensus_messages *curr;
    struct Consensus_messages *old_timestamp;
```c
struct consensus_timestamp *temp;
struct consensus_timestamp *max_conse_table_timestamp();
struct Consensus_messages *pro_max_round();

curr=conse_frontqueue;

while (curr != NULL) {
    temp=max_conse_table_timestamp(&curr->Consensus_table);
    result=conse_timestampcmp(timestamp, temp);
    if (result == 0) {
        curr = pro_max_round(curr);
        break;
    }
    else {
        if (result == 1) {
            old_timestamp = curr;
            curr = curr->next;
            delete_consemess(old_timestamp);
        }
        else {
            curr=curr->next;
        }
    }
}

return(curr);
```
struct Consensus_messages *pro_max_round(messfirst)
struct Consensus_messages *messfirst;
{
    int result;
    int theid;
    int currid;
    int currround;
    int maxround;
    int firstmaxround;
    int foundal;
    struct Consensus_messages *curr;
    static struct Consensus_messages *messmaxround;
    struct Consensus_messages *old_round;
    struct consensus_timestamp *temptime;
    struct consensus_timestamp thetimestamp;
theid=messfirst->Consensus_table.id;
maxround=messfirst->Consensus_table.BCR[theid].round;
firstmaxround=maxround;
memcpy(&thetimestamp, &messfirst->Consensus_table.BCR[theid].timestamp,
    sizeof(struct consensus_timestamp));
curr=theid;
currround=maxround;
curr=messfirst;
messmaxround=messfirst;

while ( ccurr != NULL ) {
    temptime=max_conse_table_timestamp(&curr->Consensus_table);
    result=conse_timestampcmp(&thetimestamp, temptime);
    currid=curr->Consensus_table.id;
    currround=curr->Consensus_table.BCR[currid].round;
    if ( (result == 0) && (currid == theid) && (currround>maxround) ) {
        maxround=currround;
    }
    curr=curr->next;
}

if ( firstmaxround < maxround ) {

currid=theid;
curr=messfirst;
messmaxround=messfirst;
currid=curr->Consensus_table.id;
currround=curr->Consensus_table.BCR[currid].round;
foundal=FALSE;
while ( curr != NULL ) {
    temptime=max_conse_table_timestamp(&curr->Consensus_table);
    result=conse_timestampcmp(&thetimestamp, temptime);
    currid=curr->Consensus_table.id;
    currround=curr->Consensus_table.BCR[currid].round;
    if ( (result == 0) && (currid == theid) && (currround<maxround) ) {
        old_round=curr;
        curr=curr->next;
        delete_consemess(old_round);
    }
    else {
        if ( (result==0)&&(currid==theid)&&(currround==maxround) ) {
            if ( foundal == FALSE ) {
                messmaxround=curr;
                foundal=TRUE;
                curr=curr->next;
            }
        }
        else {

        }
    }
}
currr=curr->next;
}
}
else {
    curr=curr->next;
}
}
return(messmaxround);
}
else {
    return(messfirst);
}
}

/* ................................................................. */

/* This procedure finds the element in the 
   table with the greatest timestamp */

struct consensus_timestamp *max_conse_table_timestamp(conse_table)
struct Consensus_array *conse_table;
{

int index;
static struct consensus_timestamp temp;

index = conse_table->id;
memcpy(&temp, &conse_table->BCR[index].timestamp,
     sizeof(struct consensus_timestamp));
return(&temp);

int conse_timestampcmp(time1, time2)
  struct consensus_timestamp *time1;
  struct consensus_timestamp *time2;
{
  /* conse_timestampcmp works like the C function strcmp:
     0 if time1 = time2
     -1 if time1 < time2
     1 if time1 > time2
  */

  /* ............................................................ */
if (time1->Opnum > time2->Opnum) return(1);
else if (time1->Opnum < time2->Opnum) return (-1);

if (time1->Bitnum > time2->Bitnum) return(1);
else if (time1->Opnum < time2->Opnum) return (-1);

return(0);

/* This is the exchange routine used
by BITCONSENSUS
*/

int conse_exchange(Myarray)
struct Consensus_array *Myarray;
{
    int mayo;
    int nresp;
    int Counter=0;
    int Done=FALSE;
int Exit=FALSE;
static int replies[NUMPRO];

struct Consensus_array Otherarray;

mayo=( (int)(NUMPRO/2) ) + 1;
while ( !Done ) {
    nresp=fbcast_l("x", gaddr_p, GET_CONSEMESS, "%T[1]", Myarray, mayo,
                   "/.d", replies);

    Exit=FALSE;
    while ( !Exit ) {
        conse_receive(&Otherarray, &Myarray->BCR[My_id].timestamp);

        if (Conseless(&Otherarray, Myarray)) {
            /* don't do anything */
        }
        else if ( Conseequal(&Otherarray, Myarray)) {
            printf("Conseequal\n");
            Counter++;
            if (Counter >= NUMPRO - Numfail) {
                Exit=TRUE;
            }
        }
    }
}
else {
Conseupdate(Myarray, &otherarray);
Exit=TRUE;

if ( Counter >= NUMPRO - Numfail) {
    Done=TRUE;
}

return;

/* .............................................................................................................................. */

/* Procedures Conseless, Conseequal, 
and Conseupdate are used by 
procedure conse_exchange to compare 
tables an update 
the table that was passed to it */

int Conseless(array1, array2)
struct Consensus_array *array1, *array2;
{
    int index;
    int cmpres;
int lowerround;

for (index=0; index<NUMPRO; index++) {
    lowerround=TRUE;
    cmpres=conse_timestampcmp(&array1->BCR[index].timestamp,
                              &array2->BCR[index].timestamp);
    if ( array1->BCR[index].round >= array2->BCR[index].round) {
        lowerround=FALSE;
    }
    if ( (cmpres == 1) || ( (cmpres == 0)&&(lowerround==FALSE) ) ) {
        return(FALSE);
    }
}
return(TRUE);
}

int Conseequal(array1, array2)
struct Consensus_array *array1, *array2;
{
    int index;
    int cmpres;
    int equalround;
for (index=0; index<NUMPRO; index++) {
    equalround=TRUE;
    cmpres=conse_timestampcmp(&array1->BCR[index].timestamp,
                             &array2->BCR[index].timestamp);
    if (array1->BCR[index].round != array2->BCR[index].round) {
        equalround=FALSE;
    }
    if ( (cmpres != 0) || ( (cmpres==0)&&(equalround==FALSE) ) ) {
        return(FALSE);
    }
    return(TRUE);
}
Conseupdate(array1, array2)
struct Consensus_array *array1, *array2;
{
    int index;
    int cmpres;
    int higherround;

    for (index=0; index<NUMPRO; index++) {
        higherround=FALSE;
    }
}
cmpres = conse_timestampcmp(&array1->BCR[index].timestamp, 
                        &array2->BCR[index].timestamp);

if (array1->BCR[index].round < array2->BCR[index].round) {
    higherround = TRUE;
}

if (cmpres == -1 || (cmpres == 0) && (higherround == TRUE)) {
    memcpy(&array1->BCR[index], &array2->BCR[index],
           sizeof(struct Myconsensus));
}

return;
}

/* .............................................................. */

/* This procedure receives the messages 
orginated by the BITCONSENSUS procedure. */

void get_consemess(msg_p)
message *msg_p;
{ static struct Consensus_array temp;
    int index;
    msg_get(msg_p, "%T[i]", &temp);
    reply(msg_p, "%d", My_id);
    index=temp.id;
    queue_consemess(&temp);
}

/* This procedure implements the consensus protocol */

int Bitconsensus(Opnum, Bitnum, val)

int Opnum, Bitnum, val;
{
    int j;
    int round;
    int Maxround;
    int I_am_a_leader;
    int single_leader;
int count_leaders;
int ahead_agree;
int leader_agree;
int leader_val;
int other_leader_val;
struct Consensus_array Consensus;

for(j=0;j< NUMPRO; j++) {
    Consensus.BCR[j].round=0;
    Consensus.BCR[j].sugval=0;
    memset(&Consensus.BCR[j].timestamp, 0,
           sizeof(struct consensus_timestamp));
}

round=0;
while (TRUE) {
    round++;
    Consensus.id=My_id;
    Consensus.BCR[My_id].round=round;
    Consensus.BCR[My_id].sugval=val;
    Consensus.BCR[My_id].timestamp.0pnum=0pnum;
    Consensus.BCR[My_id].timestamp.Bitnum=Bitnum;

    conse_exchange(&Consensus);
    MAXround=0;
for(j=0; j< NUMPRO; j++) {
    if (Consensus.BCR[j].round > Maxround) {
        Maxround = Consensus.BCR[j].round;
    }
}

for(j =0; j <NUMPR0; j ++) {
    if (Consensus.BCR[j].round == Maxround) {
        leader_val=Consensus.BCR[j].sugval;
        break;
    }
}

if (round == Maxround) {
    I_am_a_leader=TRUE;
}

if (I_am_a_leader) {
    count_leaders=0;
    for(j=0; j< NUMPR0; j++) {
        if (Consensus.BCR[j].round == Maxround) {
            count_leaders++;
        }
    }
    single_leader= (count_leaders==1);
    if (single_leader) {
        ahead_agree=TRUE;
    }
for(j=0;j<NUMPRO;j++) {
    if ( (Consensus.BCR[j].sugval != val) &&
        (Consensus.BCR[j].round >= Maxround-1) ) {
        ahead_agree=FALSE;
    }
}

if ( ahead_agree ) return( val );

else {
    leader_agree=TRUE;
    for(j=0;j<NUMPRO;j++) {
        if ( (Consensus.BCR[j].sugval != val) &&
            (Consensus.BCR[j].round == Maxround ) ) {
            leader_agree=FALSE;
        }
    }
    ahead_agree=TRUE;
    for(j=0;j<NUMPRO;j++) {
        if ( (Consensus.BCR[j].sugval != val) &&
            (Consensus.BCR[j].round >= Maxround-1) ) {
            ahead_agree=FALSE;
        }
    }
    if ( leader_agree && ahead_agree ) return( val );
}

if ( ahead_agree ) return( val );
if ( !leader_agree )
    val=Flip_Global_Coin(Opnum, Bitnum, round);
}

else {
    leader_agree=TRUE;
    for(j=0;j<NUMPROC;j++) {
        if ( (Consensus.BCR[j].round == Maxround) &&
            (leader_val != Consensus.BCR[j].sugval) ) {
            leader_agree=FALSE;
            break;
        }
    }

    if ( leader_agree ) {
        val=leader_val;
    }

    else {
        val=Flip_Global_Coin(Opnum, Bitnum, round);
    }
}

/*............................*/
void test_loop()
{
    char buffer[10];
    int Opnum = 1;
    int Bitnum = 1;
    int myval;
    int consers;
    int j;
    void group_change();

    /* Make the process join the group */

gaddr_p = pg_join("coinservice", PG_MONITOR, group_change, 0, 0);

    if (addr_isnull(gaddr_p)) {
        printf("Couldn’t join group...\n");
        exit(1);
    }

    printf("My ID: %d\n",My_id);
isis_start_done();
isis_accept_events(0);

myval=My_id % 2;

printf("myval : %d\n", myval);
consres=Bitconsensus(Opnum,Bitnum,myval);
printf("The consensus result is: %d\n",consres);
myval=(My_id+1)%2;
consres=Bitconsensus(Opnum,Bitnum,myval);
printf("The consensus result is: %d\n",consres);
exit(0);
}