1996

Virtual Central Control.

Raghuram A. Yedatore
Louisiana State University and Agricultural & Mechanical College

INFORMATION TO USERS

This manuscript has been reproduced 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.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

VIRTUAL CENTRAL CONTROL

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by
R. Kuram A. Yedatore
B.E. in Computer Science and Engineering, University of Mysore, 1987
M.S. in Systems Science, Louisiana State University, 1996
December 1996

To my parents, Radha and Ananth

Acknowledgments

First and foremost, I sincerely thank my advisor and guide, Prof. S.S. Iyengar, without whom this would not have been possible. He has been kind, patient and understanding throughout my stay at LSU. From him, I have learnt what one can achieve through perseverance. I am indebted forever to my dear friend and philosopher, Dr. Amit Nanavati. He has been instrumental in this work – his suggestions have had tremendous impact. With him I discussed endlessly every step of my research, and in the process realized how important every step is in a journey that is a 1000 steps long. He also taught me to appreciate Indian classical music. A very special thanks to Dr. Vinayak Hegde (hedju), who instigated all this by putting the thought of doing a Ph.D in my head. I would like to thank the Department of Computer Science for supporting me for 5 years. Thanks are due to Dr. Terry Geske for his support in providing the much needed financial help during my second year at LSU.

I would also like to thank Prof. Donald Kraft, Prof. Bush Jones, Prof. Kwei Tang and Prof. Gary Sanger who took time from their busy schedules and agreed to be on my committee.

I thank Prof. Ken Birman and Prof. Fred Schneider of Cornell University for their help during problem formulation.

I will always remember Dr. Mohan Sharma for all his help during my first semester. I owe a lot to Cindy and Rajen as I can count on them whenever I need them. Dr. Sankar (koo) for pushing me to work all along, although he did not succeed. He had no choice but to grudgingly teach me Tamil. Dr. Ramana Rao

for showing me easy steps to get a Ph.D that never worked for me. Dr. Chandra Shrivastava for reminding me everyday that I was enrolled in a doctoral program. She was always ready to give a pep-talk which I managed to avoid every time. Sundar, the uncool uncle, who showed me how to stay active all the time. I am still working on it but nowhere close to him. He was by my side during so many troubled times. I would like to thank my roommates Sai, Aravind and Giri who made my stay very memorable. Giri, for all the 'rock-n-roll'. To my Katte, a home away from home. My childhood friend Chandru, the years I spent with him will always be cherished. To Shivi, who was always willing to listen. And to Doc., Phatak, Harish-Anita, Dr. LP, Dr. Daryl Thomas, Murali, Babu, Utpal for all the fun times.

Last but not the least, my family members who have encouraged and supported me throughout my studies. To my cousin Ravi, who, under no obligation was always obliging in taking care of the mundane jobs that I forced on him. To my cousin C.V. Raghavan, who helped in applying to graduate schools in the U.S. To my uncle Somashekar, for his willingness to support me during my initial stay. My parents, about whom I have so much to say, but don't know where to start and where to end.

This research is partially funded by ONR Grant No. N00014-94-1-0343 to Prof. Iyengar.

Table of Contents

List of Figures

Abstract

A distributed system is a set of cooperating computers (processes) communicating with each other to achieve a common goal. They are broadly classified as centralized and decentralized systems. In a centralized system, a single computer plays the role of central coordinator and controls *all* the system activities. Whereas in a decentralized system, all the cooperating processes have an equal role to play, therefore solutions to problems involve instructing all the processes and coordinating their actions.

Although a centralized system facilitates program development, it has serious drawbacks. If the coordinator fails, the system effectively breaks down. Also, the coordinator can become a performance bottleneck. On the other hand, a decentralized system does not suffer from these shortcomings, but program development is more difficult.

In this research, we develop a paradigm for a distributed system that provides the view of a centralized system even though the underlying system is decentralized. Special processes called *agents* reside on all participating computers. One of the agents acts as a leader and coordinates activities of other agents. Agents communicate through a fault-tolerant *agent-to-agent* protocol. The paradigm defines a *one-shot* computation, a construct that enables the expression of programs in a *simulated* centralized environment, thus making program development easier. Programs, written in terms of one-shot computations that are encapsulated in *templates* which hide all the lower level details from the programmer. We show that every computation can be expressed as a one-shot computation.

The proposed paradigm has a high degree of fault-tolerance. It tolerates up to $n-1$ failures in a system of n processors. A prototype has been implemented using PVM.

Chapter 1

Introduction

The area of distributed processing has been the focus of a number of studies over the past decades. Some of the problems involving modeling of distributed systems are still being actively pursued, for they form the cornerstones of more recent developments. In recent years, the proliferation of computer networks have made resource sharing (hardware, software and data) very effective. When the work load on one computer increases, it has to be shared among other computers. The idea of resource sharing has been in existence for quite some time and has proven very effective in the last two decades. Resource sharing motivates solutions for some of the basic distributed processing problems. We briefly discuss some of the models in this area and the contributions of this dissertation. Although there is no consensus on the definition of a distributed system, an environment in which the computers and resources are connected using a network with the objective of sharing resources is termed as a distributed system (Figure 1.1). Also, a distributed system is nothing but a collection of computers interconnected by a network, that are collaborating to achieve a common goal. Mainly, distributed systems are characterized by:

- no global clock.
- no global memory.

Essentially, every component in a distributed system is independent. If one of the computers go down, the system doesn't necessarily come to a halt. This is an inherent advantage due to the structure of a distributed system. The main goal

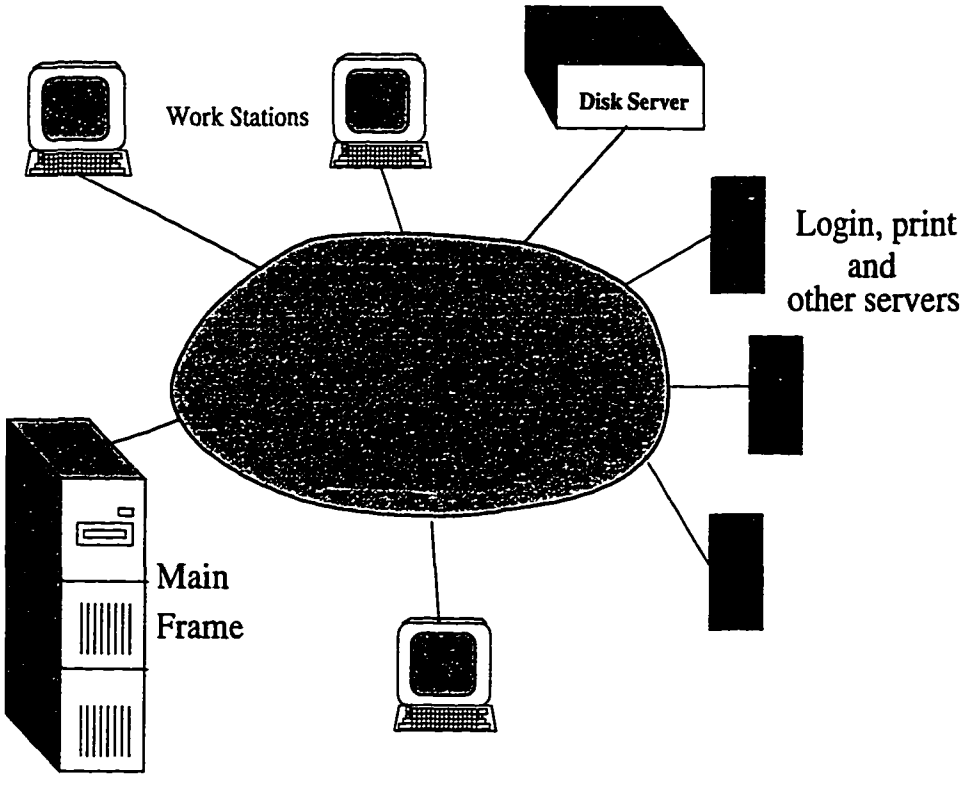

Figure 1.1: A Typical Distributed System

behind building a distributed system is that the entire system should appear as a single system to the end user.

1.1 Characterization of Distributed Systems

How to evaluate a distributed system ? What to look for in a distributed system ? Six characteristics[4] have been identified to measure the usefulness of distributed systems.

- *resource sharing*
- *openness*
- *concurrency*
- *scalability*
- *fault tolerance*

A resource can be anything from a simple data file to an expensive hardware. Even software is a resource. When computers are connected through a network, but for the basic operating system, there is no need for all the computers to have compilers, application programs, etc. It suffices if only a subset of those store copies of the software. Should one of the computers go down, the software this computer has will be available on some other computer on the network. In fact, this idea is made use of in replicated databases. Any process or computer that shares it's resource or offers services is called a *server*, for example, the *Disk Server* in Figure 1.1. A process or a computer that requests or uses these services is called a *client*. The *client-server* model is used very widely.

As a distributed system is made up of a collection of computers, it is feasible to assume that the computers and peripherals are added and/or removed from the system. The design of the system should allow modifications (addition of services)

to the system without disrupting its operation. *Openness* is defined by the degree to which such modifications can be done. In simple words, the system should be easily extensible. They may be extended by adding hardware and software from different vendors. All the vendors should adhere to pre-defined standards for interfacing with other systems. For example, *network file system* (NFS) available on all the UNIX platforms allow sharing of *directories*.

Concurrency is important for efficient operation of the system. It is more than one thing happening simultaneously. Each computer in a distributed system will have processes running concurrently (in parallel) on each of them. Many servers offering services will be responding to different requests from clients simultaneously. Concurrency is natural in distributed systems because actions of each component is independent.

Distributed systems are effective when they are scalable due to ever increasing need for more computation power, sharing etc. Some of the systems have been designed to work in a LAN environment so they can have a maximum of few hundreds of workstations, for example, NFS. A well known system that can handle connections of the order thousands is the *Andrew file system* (AFS) developed at the Carnegie Mellon University. The primary goal of both AFS and NFS is to allow heterogenous platforms to share their files. There should not be any need to change existing software, both system and application, when the system has to be scaled.

The characteristics discussed so far are so attractive, the question now is how does the system behave if its components fail. This is an important design consideration. Despite component failures, the system should continue to work. There are two approaches to fault-tolerant design[4]:

- *hardware redundancy*: the use redundant components.
- *software recovery*: the design of programs to recover from faults.

Hardware redundancy guarantees the availability of a service even when one of the computers go down. For example, a database may be replicated on several computers to ensure the data availability. Software recovery involves recovery of data to the state it was before the failure. Whatever the underlying mechanisms are, the user should not be able to observe failures or recoveries. *Process groups* paradigm, discussed in detail in the next chapter, provides mechanisms that are useful in handling faults. ISIS [5, 6, 18],a reliable distributed software environment makes use of this paradigm heavily. *Process groups* are used in this research also.

Transparency is to hide from the user the collection of components in a distributed system. It should feel as one whole system. An user or application programmer don't have to know the details of the system if his interest is to make use of a service provided by the system. As an example, in a mulitprocessor system, the processor on which a given process runs is transparent to the user. The *Advanced Network System Architecture* [ANSA 1989] and *International Standards Organization* [ISO 1992] have identified eight forms of transparency[19]. They are:

- *Access transparency* allows access of information on local and remote sites using identical operations.
- *Location transparency* enables access of information without knowledge of the location.
- *Concurrency transparency* allows several processes to concurrently access shared information without interference between them.
- *Replication transparency* allows multiple copies of information to be used to increase reliability and response time without knowledge of the replicas to users or application programs.

- *Failure transparency* hides failures of software and hardware components thus allowing users to perform their.
- *Migration transparency* allows movement of information from one component (computer) to the other without affecting the user operations.
- *Performance transparency* allows reconfiguration of the system to improve performance.
- *Scaling transparency* allows to the system to scale gracefully without change to the system structure.

1.2 Classification of Distributed Systems

Distributed systems can be classified in two different ways. The first type is based on timing constraints. Here, assumptions are made on the bounds on process execution speeds, and/or communication delays. Such system are called synchronous. These assumptions are not made in asynchronous systems, processes are allowed to execute at their own speed. Also, no restriction is placed on communication delays.

The second type of classification is based on the control. Systems where control is centralized are called centralized systems. There is a central process that controls or makes decisions for the entire system. In a decentralized system, control is divided more or less equally among all the processes.

1.2.1 Centralized Systems

In a centralized system, see Figure 1.2, one computer plays the role of central coordinator or agent. Programs are scheduled for execution by the central agent, a request for a resource, say a printer, must be sent to this computer. Any activity in

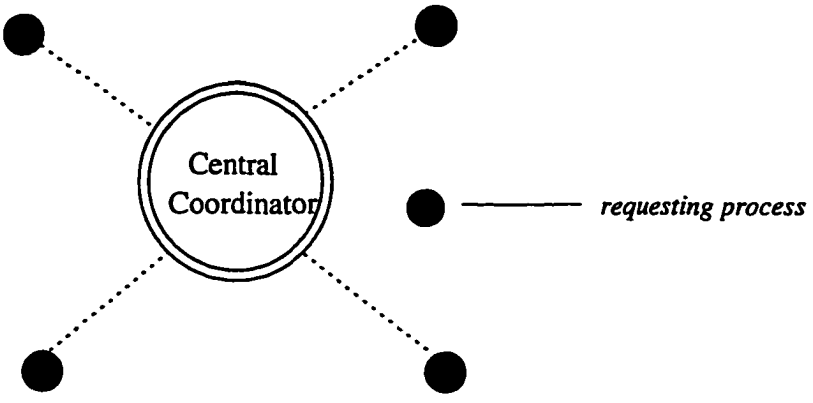

Figure 1.2: Centralized Control

general, is routed through the central agent. It is connected to an underlying network. Other computers (typically workstations) are also connected to the network. The workstations request the coordinator for their needs, for e.g., to print a file, for a copy of a compiler or editor, and so on. These computers are typically located in the same building. The main disadvantage of a centralized system is the central agent is the bottleneck. If it crashes, the entire system is down. As the other computers depend on the central agent completely for their interaction with the outside world, they just have to wait for the centralized agent to come up again. Also, If the workload increases, the performance degrades. Thus such a system cannot scale.

1.2.2 Decentralized Systems

In a decentralized system, geographically separated computers are connected by a local area network or a wide area network. These computers communicate with each other to achieve a common goal. Communication links may severe, one of the computers may crash and so on. The advantage of a distributed system despite these failures or shortcomings is, the system continues to operate by dividing the load among the computers that are still running.

Both the paradigms above offer an excellent platform to develop algorithms that are required for the ever rising demands of the applications. Developing algorithms for a centralized system is far less difficult when compared to decentralized system. For e.g., suppose that there is an expensive resource that needs to be shared and only one computer can access this resource at any given time. A computer requests this resource; once it is allocated, it uses the resource and releases it for use by other computers. This is nothing but the classical mutual exclusion problem. Several algorithms have been suggested for this problem in both the paradigms [3, 11]. It is obvious that in a centralized environment, the computers that are in need of the resource should request the central agent. The central agent does the job of scheduling the resource. The scenario is very different in case of a decentralized system as there is no central agent or coordinator. The computers should communicate with one another and schedule the resource. It is apparent that the same problem becomes much harder in a decentralized environment. So it is very important to model distributed computing environments that ease the development of applications in such an environment.

1.2.3 Synchronous and Asynchronous Systems

Every system is asynchronous. Synchronous systems in which the processes run in lock step to satisfy the definition of asynchronous systems as well. In other words, synchronous systems are a special case of asynchronous systems. Since no assumptions are made in case of asynchronous systems, a protocol designed for an asynchronous system can be used in any distributed system and not the other way. As in centralized systems, development of algorithms for synchronous systems is a lot easier than their asynchronous counterparts. Techniques like timeouts and time-

based protocols can be used only when a system is synchronous. For instance, a process will only wait till a pre-specified amount of time for a message from another process and proceeds whether if it receives or not. This provides a mechanism for failure detection. This simplifies the design of distributed algorithms.

An asynchronous system is the weakest possible model in distributed systems. As mentioned earlier, there exist no bounds on processor speeds, message delivery delays, or the time necessary to execute a step. Also, there are no synchronized clocks or reasoning based on global real-time. Communication remains the only possible mechanism for synchronization in such systems [4]. In practice, the sources of asynchrony are variable and unexpected workloads. Thus asynchronous systems are very practical. Many distributed algorithms require that the messages be ordered. For e.g., If a set of processes are maintaining replicas of a database, these replicas should be updated in the same order inorder to maintain the consistency. An algorithm for this problem in an asynchronous environment is harder than the one in a synchronous environment.

1.3 Motivation

Solutions to well known problems in the area of distributed systems are somewhat less difficult in a system with centralized control. For instance, problems such as mutual exclusion, consensus, broadcast, multicast, and concurrency control are harder to solve in an environment with decentralized control because of increased message passing and bookkeeping.

As an illustration, consider the problem of consensus in a distributed system. This problem requires agreement among a set of processes (or equivalently, comput-

ers) on a common decision. In a centralized system, this problem is solved by the centralized agent which collects the (possibly different) individual intermediate decisions of the cooperating processes and achieves eventual consensus. On the other hand, in a decentralized environment, the cooperating processes communicate with each other and reach a consensus, possibly after several 'rounds' of negotiations. Reaching consensus may be further complicated by a process repeatedly changing its decision based on messages received. Clearly, in such a situation, the algorithm for achieving consensus will be more complicated and in general, may require more messages. Several algorithms for solving the consensus problem in a decentralized environment exist [11, 8, 15].

As a different example, consider the problem of ordering of *broadcast* messages received by a process. In a broadcast operation, the originator sends a message to each of the other processes in the system. Broadcast messages are usually *concurrent*, i.e. many processes may broadcast their messages simultaneously. Many applications(e.g. database) require that messages be ordered to maintain a consistent state. In a centralized system, a process that intends to issue a broadcast message communicates with the centralized agent which then broadcasts this message to each of the other processes on behalf of the process in question. The presence of a centralized agent greatly simplifies the broadcast problem in a centralized environment. Since all broadcast messages are routed through the CA, they are ordered automatically. Kaashoek et al.[14] proposed a protocol for the broadcast problem in a centralized environment.

In a decentralized system, each process keeps track of the messages it receives from the other processes and ordering these messages may involve the exchange of

several rounds of messages among the processes. Most algorithms use the technique in [13] to solve the problem.

In a centralized environment, solutions to the problems cited above rely on the centralized agent to a large extent. Further, there is an underlying similarity between the different solutions due to the presence of this centralized agent. While the centralized agent simplifies solutions, the approach has some serious drawbacks. Firstly, there is the danger of breakdown due to failure. If the centralized agent fails, the system breaks down. Secondly, since all bookkeeping is handled by the centralized agent, there is a significant bottleneck at this agent which leads to performance degradation. In contrast to this, a different approach may be needed to solve each problem in a decentralized system. A process involved in a distributed system should, in general, keep track of every other process.

Our approach is motivated by the simplicity offered by the centralized agent in a centralized environment. We want to 'mimic' this agent in the more generalized setting of decentralized systems. To achieve this, processes involved are given the illusion of a centralized framework through a *virtual* centralized agent. We say 'virtual' since no single fixed centralized agent exists in the framework. Chapter 3 describes this architecture in detail. The most important advantage comes from the fact that it provides the virtues of a central agent in a decentralized environment, thereby eliminating the constraints of centralized environment and the difficulties of decentralized environment.

1.4 Dissertation Outline

The presence of a central coordinator (agent) in a centralized environment reduces the burden on the programmer and also facilitates the development of efficient algorithms. The fact that all the processes need to know and communicate only with the central agent is attractive. Despite this salient feature, this paradigm has serious shortcomings. Whereas in a decentralized environment, even though the bottleneck due to central control in a centralized system is eliminated, the development of algorithms is more difficult. Therefore the development of an architecture which exploits the advantages of both the paradigms while eliminating their limitations is worth considering.

In this research we model an architecture called *Virtual Central Control* (VCC). Agents on every computer act as central coordinators for processes on that computer. Each process in the system 'thinks' that there is a central agent that services its requests. The central agent here is virtual, which means that there is no fixed single centralized agent. This approach has the following advantages:

1. provides a centralized view
2. work get divided among all the agents
3. it is fault-tolerant.
4. offers a template based framework that hides all the lower-level details from an application programmer.

Chapter 2 describes briefly other works that is closely related our research. VCC architecture is discussed in Chapter 3. Fault-tolerance supported by VCC is dis-

cussed in Chapter 4. A prototype implementation on PVM [22] and is explained in Chapter 5. Possible extensions to VCC and a summary is presented in Chapter 6.

Chapter 2

Preliminaries and Related Work

2.1 Preliminaries

This Chapter is divided into two parts. In the first, we briefly describe some basics of distributed systems. Specifically, we concentrate on those aspects that are relevant to our research. In the second part, we discuss other works closely related to our research.

2.1.1 Distributed Algorithms

In a distributed system, a set of processes, executing on the same or on different computers, cooperate to achieve a common goal. Often, the same algorithm is executed by all the processes involved except for e.g., the central agent in a centralized system. The processes communicate by exchanging messages. The type of messages exchanged by the processes depends on the state of the algorithm. These messages have predefined meaning and the algorithm dictates the action to be taken based on the type of the message received. The set of messages coupled with their predefined meaning(s) constitutes what is formally called a *protocol*.

It is possible that one of the processes may fail during the execution of an algorithm. If the algorithm is robust, other processes detect that a process has failed and continue to solve the problem in the absence of the failed process. An algorithm that runs despite failures is called fault-tolerant.

2.1.2 Ordering of Events

In a distributed system, processes are spatially separated and the events occur asynchronously. For e.g., two different processes may issue update operations in a database application simultaneously. As distributed databases maintain replicas for the purpose of fault tolerance, response time, etc., these updates should happen in the same order on all the replicas to ensure consistency.

There are innumerable distributed applications where event ordering is crucial. The absence of a global clock makes ordering difficult. As Birman puts it, "Each machine has its own clock, and clock synchronizations is at best imprecise in distributed systems". Sending and receiving a message by a process are events. Two processes may send messages concurrently. How can these events be ordered ? Leslie Lamport answers the question in his seminal paper [13]. He defines the "happened before" relation and the notion of logical time. The "happened before" relation, denoted by, " $\rightarrow$", is only a partial ordering of the events in the system.

Definition of happened before relation: The relation on the set of events of a system is the smallest relation satisfying the following three conditions:

1. If a and b are events in the same process, and a comes before b, then $a \rightarrow b$.

2. If a is the sending of a message by one process and b is the receipt of the same message by another process, then $a \rightarrow b$.

3. If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$.

Two distinct events a and b are said to be *concurrent* if $a \not\rightarrow b$ and $b \not\rightarrow a$.

$a \rightarrow b$ can also be viewed as event a causally precedes event b. Two events are concurrent if neither can causally affect the other.

Logical clocks: In asynchronous distributed systems there is no global real-time clock. But the events must be ordered some how. Lamport suggested that each process maintains a local variable LC called its *logical clock.* Initially all processes set their LC to zero. When a process sends a message m, it is timestamped TS(m) with the current value of LC. LC is modified with each event e occurring in a process as follows:

$$LC(e) = \begin{cases} LC + 1 & \text{if } e \text{ is an internal or send event} \\ max\{LC, TS(m)\} + 1 & \text{if } e = \textit{receive(m)} \end{cases}$$

In other words, when a receive event is executed, the logical clock is updated to be greater than both the previous local value and the timestamp of the incoming message. When an internal event or send event is executed, the logical clock is simply incremented.

2.1.3 Vector Time

Vector times were proposed by Fidge and Mattern[16]. They extend logical clocks discussed in the previous section to a decentralized system with n processes. A vector time for a process p_i, denoted $VT(p_i)$, is a vector of length n (where $n = |P|$), indexed by process-id.

1. When p_i starts execution, $VT(p_i)$ is initialzed to zeros.

2. When a process p_i sends a message m, $VT(p_i)[i]$ is incremented by 1.

3. When process p_j delivers a message m from p_i containing $VT(m)$, p_j modifies its vector clock in the following manner:

$$\forall k \in 1 \ldots n : VT(p_j)[k] = max(VT(p_j)[k], VT(m)[k]).$$

That is, the vector timestamp assigned to a message m counts the number of messages, on a per-sender basis, that causally precede m.

Rules for comparing vector timestamps are

1. $VT_1 \leq VT_2$ iff $\forall\, i : VT_1[i] \leq VT_2[i]$

2. $VT_1 < VT_2$ if $VT_1 \leq VT_2$ and $\exists\, i : VT_1[i] < VT_2[i]$

Also, given messages m and m', $m \rightarrow m'$ iff $VT(m) < VT(m')$, meaning vector timestamps represent causality precisely.

2.2 Related Work

Considerable amount of work has been done to model distributed systems. Each work address different aspects. ISIS[5, 6, 16, 18], a frame work for developing reliable distributed software, gives the illusion of a synchrony in an asynchronous environment is described below. FrameWorks[20] is a system for writing applications to run in a distributed environment. It provides an array of templates with different behaviors from which an user can chose to write an application. Also, FrameWorks takes advantage of idle workstations by scheduling user processes to run on them. In [2], an infrastructure to develop large scale distributed applications called *Distributed Applications Framework* (DAF) is discussed. DAF defines a system as a set of services offered by a collection of servers over the network. Application programmers use these services to build applications.

Process groups paradigm is discussed in detail. Both ISIS and VCC use this notion.

2.3 ISIS

A decentralized environment poses challenges to the development of algorithms. Researchers have invented clever tricks to solve quite intriguing problems. In a synchronous system, one event occurs at a time so it is relatively easy to develop algorithms in such an environment. The generality in asynchronous systems comes from allowing heterogeneous platforms to coexist and participate in problem solving. Since no such assumptions can be made in an asynchronous environment, synchronous solutions, in general, are not always easily extensible to asynchronous ones.

It is hard to write a robust distributed application as the algorithms in the literature make assumptions such as synchronous environment or ordering of messages, FIFO, Causality, etc. Asynchronous propagation of information among processes is what makes the development hard.

Two aspects of ISIS are key to its overall approach [16]:

- An implementation of *virtually synchronous process groups.* Such a group consists of a set of processes cooperating to execute a distributed algorithm.

- A collection of reliable multicast protocols with which processes and group members interact with groups.

ISIS has introduced the notion of *virtual synchrony* and *process groups.* The idea is to give the view that the underlying system is synchronous as it eases the

burden on the programmer. As Birman puts it, "...involves a complex synchronization algorithm, probably beyond the ability of a typical distributed applications programmer." The virtual synchrony mechanism gives the user an illusion of a synchronous system. Process groups are nothing but a collection of processes not necessarily on the same computer.

2.3.1 Virtual Synchrony

ISIS is a *virtually synchronous* programming environment. Intuitively, this means that users can program as if the system scheduled one distributed event at a time. A system is called *synchronous* if it were to behave this way actually; such an environment greatly simplifies the development of distributed algorithms but offers little opportunity to exploit concurrency. The "schedule" used by ISIS is, however, synchronous in appearance only. This means, algorithms can still be developed and reasoned about using a simple, synchronous model.

In a virtually synchronous environment, programs can be written that behave as if distributed actions were performed in lock-step. The underlying execution however is much more concurrent. For example, the update operation on a replicated database operates asynchronously. That is, the process that requested an update operation may continue without waiting. Using the tools provided by ISIS, it can be programmed as if the updates occur instantaneously. ISIS guarantees that any sequences of actions (including indirect ones) will not cause a read that is performed after such an update to be satisfied using a prior value of the updated item. As another example, two processes receiving the same multicasts see the corresponding local events in the same relative order as if the system is synchronous.

A virtually synchronous execution is thus characterized by the following property: It will appear to any observer – any process using the system – that all processes observe **the same events in the same order**. This applies not just to message delivery events, but also to failures, recoveries, and group membership change events. This enables one to make apriori assumptions about the actions other processes will take and thus simplifies algorithmic design.

2.4 Process Groups

A process group is a collection of processes and each group has a name that represents all the processes in that group. The members of a group need not be identical, nor is there a limit on the groups to which a process may belong. Members join and leave the groups dynamically. A message sent to a group will reach all its members. When multiple processes need to cooperate, they can be structured into process groups.

ISIS supports four types of groups [5, 16, 18]: *peer group, client/server group, diffusion group, hierarchical group.* The simplest of all is the *peer group* in which processes cooperate as equals in order to get a task done. They may manage replicated data, subdivide tasks, monitor one another's status, or engage in a coordinated distributed action. In the *client/server group*, a peer group of processes act as servers on behalf of a potentially large set of clients. Clients interact with the servers in a request/reply style, either by picking a favorite server or by multicasting to the whole server group. A *diffusion group* is a type of client-server group in which the servers multicast messages to the full set of servers and clients. Clients are passive and simply receive messages. Diffusion groups arise in any application that broad-

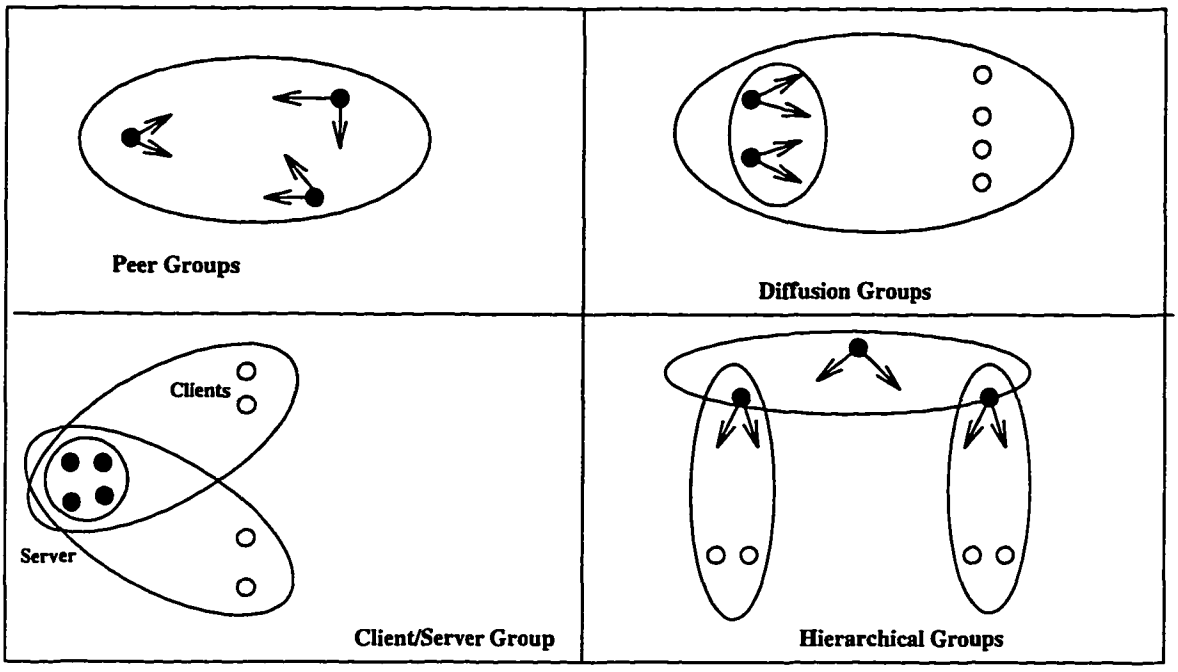

Figure 2.1: Types of process groups

casts information. Finally, *hierarchical group* structures arise when larger server groups are needed in a distributed system. Hierarchical groups are tree-structured sets of groups. Four types of groups are illustrated in figure 2.

2.4.1 Process Group View

A *process group view* (or just *view*) is a snapshot of the membership and global properties of a process group at some (logical) instant of time. A *view* of a process group is a list of its members. A *view sequence* for a process group g is a list $view_0(g), view_1(g), \ldots, view_n(g)$, where

1. $view_0(g) = 0$,

2. $\forall\, i : view_i(g) \subseteq P$, where P is the set of all processes in the system, and

3. $view_i(g)$ and $view_{i+1}(g)$ differ by the addition or subtraction of exactly one process.

Processes learn about group membership changes only through this view mechanism including the failure of other group members. A process p belonging to g, "learns" of $view_i(g)$ when it receives a message having $view_i(g)$.

2.4.2 Virtually Synchronous Tools

ISIS defines [18] and uses three communication primitives that form the backbone of the system. They are called **broadcast** primitives, *atomic broadcast (ABCAST), causal broadcast(CBCAST) and group broadcast (GBCAST).* All the broadcast primitives are *atomic*; that is, a broadcast made to a set of processes is eventually received by all operational destinations or by none, even in the presence of failures. These broadcast primitives are also referred to as multicast primitive in the literature.

ABCAST primitive: Many applications in which a number of concurrently executing processes communicate with a shared distributed resource, are sensitive to the order in which requests arrive. For example, concurrent operations on a shared replicated FIFO must be received and processed at all copies in the same order. This ordering requirement is satisfied by the *ABCAST* primitive, which delivers messages atomically and in the same order everywhere. If all requests for queue operations are transmitted using this primitive, the enqueuing operations would look synchronous relative to other such operations on the same queue.

CBCAST primitive: Lamport observed that , the ordering of events is meaningful only when information could have flowed from one to the other through some chain of message transmissions and receptions. ISIS makes use of this observation

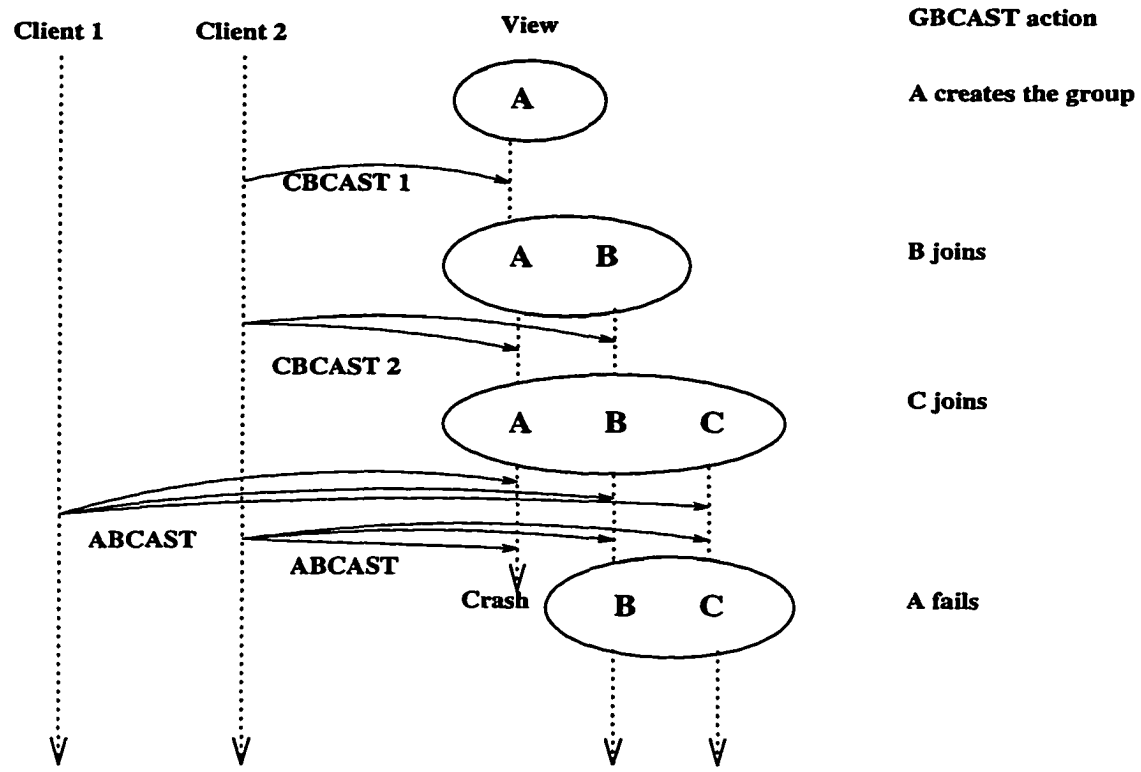

Figure 2.2: Clients communicating with a process group

to define the CBCAST primitive. Two broadcast events are said to be *potentially causally related* if information about the first could have reached the point where the second was begun before it was initiated there. This clearly implies that two broadcasts issued by a single process are always potentially causally related. The CBCAST primitive guarantees that if any invocations of CBCAST are potentially causally related, the corresponding messages are delivered everywhere in the order of invocation.

GBCAST primitive: GBCAST is used by the system to manage group addressing. It is basically used to inform operational group members when another member fails, recovers, joins, or withdraws voluntarily. All the members need to do is to maintain a local copy of the *view* (described above), updating it upon the

receipt of a GBCAST message, and acting on it directly without needing any further agreement protocols. This means that the GBCAST be relatively ordered to other events in the same way at each member.

Figure 2.2 shows an example to demonstrate the use of communication primitives. Two clients communicate with a process group using CBCAST and ABCAST. All processes see the event changes in the same order. The actions taken by GBCAST are also shown in the figure.

2.5 FrameWorks

FrameWorks is a computational model and a system for the generation of distributed applications in a network of workstations. Programs are written as sequential procedures enclosed in *templates*. The templates hide all the communication and synchronization details.

An application program consists of communicating processes. The programmer views each process as a sequential module (procedure). Modules are packaged into templates with pre-defined characteristics which can be used for specifying the scheduling/synchronization structure. Templates are used to describe the interaction of a module with other modules. Most of the information regarding a module's interaction with other modules is added thought a separate set of attribute bindings known as *template attachments*. It is the programmer's responsibility to chose appropriate templates that completely describes the behavior of a module in an application.

Each module needs up to three types of templates: *input, output and body* templates. An input template is responsible for correct scheduling and synchronization

of incoming messages. Similarly, an output template deals with the scheduling and synchronization of calls originating from the given module to other modules. A body template is used to assign additional characteristics to a module that modify the module's execution behavior in the distributed environment and it is optional.

Further, there are three types of input templates, *initial, in_pipeline* and *assimilator.* Initial templates allow no input from other processes. So only main module of the application may use this template. A process with an in_pipeline input template can act as a server to any of its input processes. Assimilator templates are used to merge the outputs of several processes.

The three types of output templates are *out_pipeline, manager* and *terminal.* Out_pipeline template allows for the output of a process to flow in a pipeline fashion to any other process. The manager template is concerned with the management and scheduling of multiple instances of the same module. A process with terminal type output template does not call any other process.

Executive and *Contractor* templates are the two types of body templates. An executive template is meant to serve as a user interface to the application and hence only main module can use it. A module with a contractor body template is repeatedly and asynchronously called by its input nodes. It means that the contractor hires *employees* to get the work done. This is useful when some processes in an application require significantly more computation than others.

Even though VCC uses templates, the type and purpose are quite different. The template used by VCC is discussed in the next chapter. Moreover, the user has got nothing to do with the templates, it is hidden from him.

2.6 Distributed Application Framework

In [2], a framework is provided for distributed application generation. There is a collection of servers offering services and a user has to use these services to build applications. A service may be a simple operation like copying a file or an online library system. To manage services in a system, DAF applies the notion of Name Space used by Distributed File Systems. Name Space, briefly, is the notion of giving each and every component in the system a unique name. Each server provides a UNIX-filesystem-like name space to name and manage available services. This results in ease of managing services, it is similar to managing files on a file system.

Remote Procedure Call (RPC) is adopted as a communication paradigm between servers and clients; each service is implemented as a RPC call. Servers are nothing but *service-executing-machines* whose tasks are mainly to manage available services, load new services, and answer clients' requests by invoking appropriate services. A server is implemented as a UNIX command. Services, on the other hand, are treated as building blocks for constructing the system. DAF does not define the semantics or syntax of a service, but only its invocation mechanism.

2.6.1 Server Name Space

Each server provides a name space to name and access available services. Like name spaces in UNIX-like filesystems, a server's name space is a tree-structured naming hierarchy, including three kinds of nodes: *directories, service nodes* (services), and *service links.*

2.6.2 Distributed Services

Each service is defined as a remote procedure call and implemented as a service library. A service library is made up of three major components:

- Service Function: Implements the service, including the code for calling the service.
- Data Conversion Functions: routines to encode/decode typed data to and from a network independent byte stream for input arguments and output results of the service function.
- Initial Function: Registers the service into server's name space.

From an application's point of view, specifying a service interface and implementing service code are two major tasks. Application programmers implement the service function. DAF provides a service compiler called ServiceGen which generates all the required encoding/decoding routines given the type of input and output arguments. It also generates a template which the user edits to place his function within the template.

If a service is not available, a user has to build his own service. There are no mechanisms that guarantee synchronization and scheduling which are crucial to a distributed application. The nice part of this architecture is that it provides a file system type of interface to manage services. For e.g., changing a *symbolic link* allows forwarding a service from one server to another.

Chapter 3

Virtual Central Control

3.1 The Architecture

As mentioned in the previous chapters, the goal of the architecture is to provide a simplified view of a distributed system to aid the programmer in program/algorithm development. The backbone of the architecture is a pool of computers, typically workstations on a LAN. A server process called an *agent* resides on every participating computer (see Figure 3.1). We refer to this collection of computers as a *Virtual Machine*. The server is nothing but a daemon process. All the agents form a group and use *group communications* to communicate as described in Chapter 2. One of the agents is designated as the *leader*, sometimes referred to as the *Central Agent*. Among other things, the leader coordinates the activities of other agents.

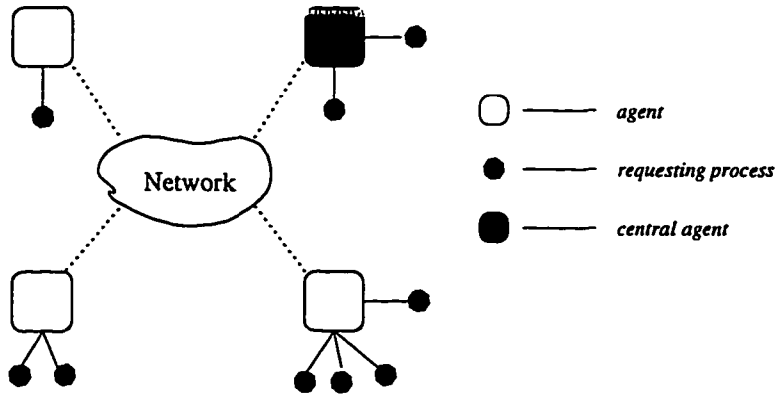

Figure 3.1: The Architecture

The agents use the Agent-to-Agent protocol (aap) to communicate. The protocol is given in fig. 3.2. The first agent creates the group and others join. Every agent at regular intervals, depending on the load, probes other agents to make sure they are alive. Note that all the agents are in a group. After a probe message is sent, the sender expects a reply within a specified timeout period. If there is no response, the agent broadcasts a *failure suspect* message to the group. It is very likely that agents may receive multiple *failure suspect* messages as well as broadcast such a message if they happen to suspect. Multiple messages are ignored and the agents go into *try once more* phase, where they try to communicate with the suspected agent once again, should they receive a reply, the suspicion is dropped and they continue as before. If no reply is received, the agent is dropped from the group.

Lemma 3.1 *The agent-to-agent protocol (aap) is correct.*

Proof: To prove aap is correct, we have to prove it satisfies the following three properties.

i) there should be no more than one leader at a time.

ii) leader is being elected.

iii) the election always yields a unique leader.

During *initialization*, agents form a group and call for a leader election. There is no leader at this time. And in the *workloop*, if the leader dies, the remaining agents confer and call for a leader election. Once again, there will be no leader till the election. Also, at any time, the group semantics ensure that the agents stop to synchronize before a leader is elected. This guarantees that the no two agents start the election independently. Thus satisfying the first property.

Every agent goes through the *Initialization* once and stays in the *Workloop* thereafter.

Initialization :

```
Join the group
If (leader is not known)
    Call for leader election
    Synchronize
    Elect a leader
else
    get the status of the group (info. about leader and other members)
```

Workloop :

```
forever do
{
  Broadcast a probe
  message=top(messagequeue)
  Case (message) of
        Agentadd: leader sends the group status to the new agent
        Agentdel: if deleted agent = leader
                        Call for leader election
                        Synchronize
                        Elect a leader
        UserPgm: Spawn the user program
        Rcvprobe: Check if all the agents replied
                        if not, Broadcast Agentdel
}
```

Figure 3.2: Agent-to-Agent protocol

Second property is trivially satisfied as all the agents synchronize before a leader is elected, in *initialization* and *workloop*.

The leader election algorithms in the literature assume unique leader ids. All the processes in a group have unique ids and any leader election algorithm will elect a unique leader. □

These collection of agents give the view of a single coordinator. Typically, in a centrally controlled system, an algorithm runs on the central coordinator and processes request the coordinator for services. In VCC, an algorithm runs on all the agents and the processes request for services with the agent residing on the same host computer. The processes are not aware of the existence of other agents. An user writes a program (or equivalently, an algorithm) with the view of central control.

The paradigm requires the user to express an algorithm in terms of three types of functions, viz., *input, output* and *computation*. Correspondingly, there are three templates. At compile time, any calls (invocation) to these functions are linked to appropriate templates. An input function is used for communication from a requesting process to the agent. An output function does just the opposite, for communication from an agent to processes. A computation function does generic computations. The semantics of these functions vary with the types of templates (section 3.4) that encapsulate them.

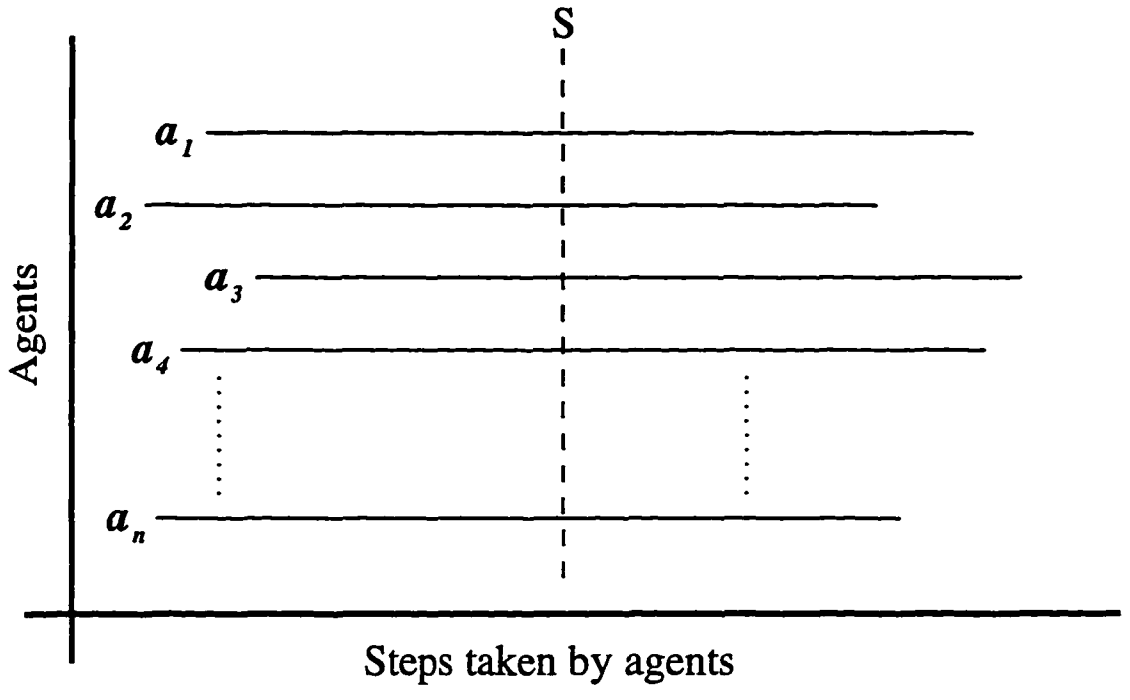

Figure 3.3: *One-shot* Computation

3.2 Events

An activity such as sending of a message, receiving of a message, or a computation step, is viewed as an event in the system. We use the notation a_i, b_j to denote event a of process i and event b of process j respectively.

3.2.1 Event Sequences

The function paradigm in VCC requires that an algorithm be expressed in terms of three types of functions *input, computation* and *output.* Also, their execution be *sequenced* in that order. We concentrate on the three types of events, execution of input, computation and output functions. The execution of an input function by an agent j is denoted by i_j. At any agent, the execution of input, computation and output is represented by the event sequence *ico.* And when we refer to more than

agent, say agents 1 and 2, the event sequence is represented by $i_1c_1o_1$ and $i_2c_2o_2$ respectively.

Let each agent's activity be described as the sequence *ico*. For any agent j, it is true that

$$i_j < c_j < o_j, \text{ where } < \text{ stands for "happens before".}$$

If we consider the *ico* sequences of m agents, $i_1c_1o_1, i_2c_2o_2, \ldots, i_mc_mo_m$, the global sequence will be an arbitrary interleaving such as

$$i_1c_1i_2o_1i_3c_2o_2i_mc_3 \ldots c_mo_mo_3.$$

Note that for any j, $i_j < c_j < o_j$.

3.3 Problem Characterization

Definition: Let S be a set of all possible *ico* sequences satisfying $i_j < c_j < o_j$, for any j. Now consider the subset G of S, such that

$$G = \{\ s \in S : \text{for all } l, m, n,\ \ i_l < o_n \wedge c_m < o_n\ \}$$

G is a set of *ico* sequences, such that, for any j, k, there can be an arbitrary interleaving of i_j's and c_k's when $j \neq k$. But when $j = k$, $i_j < c_k$. For example, the sequence $i_1i_2c_2c_1o_1o_2 \in G$ and $i_1i_2c_2o_1c_1o_2$, $i_1c_1c_2i_2o_1o_2 \notin G$. Observe that the o's start after the all the i's and c's. Any computation that generates strings in G is termed as a *one-shot* computation.

In a one-shot computation, every agent executes the input and computation functions asynchronously and waits for other agents to do so before executing the output function. In other words, the agents synchronize after executing a computation function. The *computation function* executed by all the agents gives partial

1. Collect numbers (input) from all the processes – *input function*
2. Find the maxima – *computation function*
3. Distribute the maxima to all the processes – *output function*

Figure 3.4: An Algorithm for Maxima

results. The leader collects the partial results and executes the computation function on them to obtain the final result. This is why o's start after all the i's and c's.

Informally, one-shot computations are similar to divide-and-conquer techniques. Here, a problem can be divided into a number of subproblems and each subproblem can be solved in parallel. The partial results thus obtained can be used towards the final solution. In the maxima problem, local maximas are computed first and then they are used to compute global maxima. In problems like mutex it is hard to divide the data (requests) or to solve them in a divide-and-conquer way.

An Example

Consider the maxima problem: A set of processes $P_1, P_2, \ldots, P_n$ have numbers, $N_1, N_2, \ldots, N_n$ respectively. The problem is to find the maximum of them.

An algorithm for this problem is shown in Figure 3.4. It is expressed in terms of input, computation and output functions. It is easy to see that the algorithm generates sequences in G. All the agents collect the inputs and find the maximum of them. Now all the agents have one maxima each, local maximas. The maximum of these maximas is the global maxima. The output sequence can start after finding the global maxima. Thus the *ico* sequences generated will be in G.

Some algorithms generate *ico* sequences that are not in G but the computation is a valid one. In other words, it is not a one-shot computation. For example, consider

1. Queue the requests - *input function*
2. Pick a process from the queue - *computation function*
3. Grant permission to the picked process - *output function*
4. Go to step 1.

Figure 3.5: Mutex Algorithm

the mutex problem[12]: The allocation of a single, indivisible, nonshareable resource among *n processes*, $P_1, P_2, \ldots, P_n$. The resource could be a printer or a database that requires exclusive access in order to avoid any interference.

A solution to this problem is to pick one process among the set of processes competing for the resource and assign the resource to it. Once that process releases the resource, it is assigned to another process. An algorithm expressed in terms of *input, computation* and *output* is shown in Figure 3.5. The computation and output functions ensure that only one process will be assigned the resource - this condition is not obvious in the algorithm. The agents execute steps 1 and 2, generating an arbitrary sequence of i's and c's. One of the processes is granted the resource - the first o in the sequence. At this time it is possible that one of the agents might still be in step 2, computation function, i.e. it has not generated a c. Clearly, this is a valid computation and the sequence does not belong to G. We later discuss a one-shot solution to the mutex problem and show that all the computations can be expressed as one-shot computations.

3.4 Templates

Templates are the program analogs of the *functions* discussed above. A user writes his program in terms of functions that are encapsulated in templates.

The architecture provides two types of templates:

1. Synchronous mode templates

2. Asynchronous mode templates

The type of template to be used depends on whether the algorithm in question generates strings only in G or not. If it does, the synchronous mode template should be used. If not, the asynchronous mode templates should be used. In other words, algorithms that are one-shot computations (or iterations thereof) require templates for synchronous mode, for example, the algorithm for the maxima problem (see Figure 3.4). Whereas, the algorithm for the mutex problem (see Figure 3.5) needs the asynchronous mode template.

The reason for the names is that one-shot computations can be synchronized, and computations that also generate sequences not in G are asynchronous.

A template is associated with each type of function. The *input template* is shown in Figure. 3.6. The purpose of the input function is to collect the input from the participating processes. It performs all the low level operations such as broadcasting, collecting the input, unpacking the input in a form suitable for use by input function ($func *$). The collected input is pointed to by *ptr* that ($func *$) can use to assemble the input into *param*.

The computation template encapsulates a computation function (see Figure 3.7). The *in param* is the argument to ($func *$). Once the function is executed the result (partial) is sent to the leader. The next operation depends on the type. The semantics of this template is explained in the next two sections.

```
input_template ((* func), param )
{
        Local *ptr

        Broadcast send input to all the processes
        Collect the inputs and make ptr point to them
        (* func) (param, ptr)
}
```

Figure 3.6: Input Template

```
comput_template ((* func), in param, out param )
{
        Local temp

        (* func) (in param, temp)
        send temp to leader
        if (leader) {
                Collect temp of all the agents
                (* func) (temp, out param)
                broadcast out param to all agents
        }
        else {
                recv out param from leader
        }
}
```

Figure 3.7: Computation-function Template

3.4.1 Synchronous Mode Templates

As mentioned, the input function collects the input requests from requesting processes. The leader has no role in *input* and *output* computation steps. In this mode (synchronous), the agents execute a computation function (encapsulated in template) and send the result to the leader. The leader collects the (partial) results from all the agents and executes the same of function on them to get the final result. The output function is used send the result from agents to requesting processes. The execution is synchronized at every computation function in this mode of operation. With this semantics, the algorithm in Figure 3.4 will compute the maxima when encapsulated.

The computation template hides the process of collecting partial results and applying the function again. The sending of the partial result to the leader is hidden too.

Lemma 3.2 *The maxima algorithm in Figure 3.4 is a one-shot computation.*

Proof: It suffices to show that the algorithm generates sequences only in G. The input function is executed and inputs are collected. All the agents execute computation function (find the maxima) and compute maxima. At this point, event sequence has i's and c's only. The agents send their maxima to the leader. The leader collects maximas from all the agents and then computes the final maxima. It then sends the final maxima to all the agents. The agents do nothing till the leader sends the result. Now they start executing the output function thereby generating o's. The event sequence is clearly in G. □

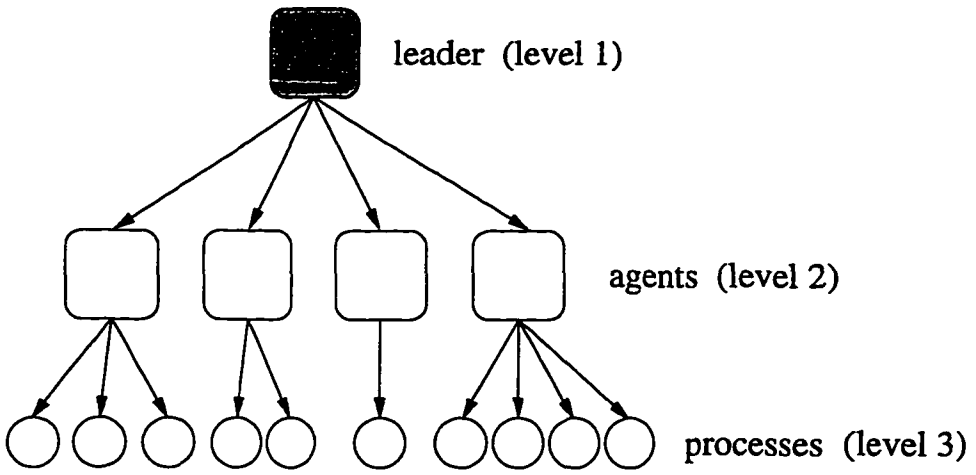

Figure 3.8: Execution Hierarchy

3.4.2 Asynchronous Mode Templates

In asynchronous mode of operation, the leader executes the entire program as opposed to the synchronous mode of operation where the leader executes only the *computation function*. The semantics of the three functions are different.

As before, all the agents except the leader collect the input agents. It can be viewed as a hierarchy of execution (see Figure 3.8). The nodes at the first and second level execute user programs. The leaves are processes that send the requests. We refer to the first level as higher level and the second as lower level. We say *What Happens at Lower Level Happens At Higher Level.*

A program runs on all processes (agents) in the first and the second level. The execution of an input function causes processes to collect inputs from processes that are one level lower. The only change in the computation function is that the leader does not automatically send the result to the agents. The semantics of an *output function* is just the opposite of input function. The processes at higher level send messages (result) to those at lower level. As the leader executes the entire program,

1. Queue the requests	- *input function*
2. Pick the process at the head of the queue	- *computation function*
3. Grant permission to the picked process	- *output function*
4. Wait for the process to return	- *input function*
5. Go to step 1.	

Figure 3.9: Mutex algorithm for Asynchronous templates

the execution history is maintained. The output function is used to pass a message (result) from the nodes at a higher level to the nodes at a lower level.

Given these semantics, the input function executed by the leader can collect input only from the agents. This implies that the leader has to wait till at least one of the agents executes a computation function. The result of this is input to the leader. Upon collecting the input, the leader executes computation function. The result gets passed on to agents when output function is executed. The agents in turn pass them to the processes. A solution to mutex is shown in Figure 3.9.

Lemma 3.3 *The Mutex algorithm in Figure 3.9 works correctly in templates for asynchronous mode.*

Proof: We show that the event sequence generated is not in G. All the agents, including the leader execute asynchronously. After queuing the input requests, the agents pick a process (computation function) which is sent to the leader. The leader queues the requests it has received so far and picks one among them. So far in the execution the event sequence has i's and c's only. The leader executes the output function and send the result (picked process) to the appropriate agent. This is the first o in the sequence. The agent upon receiving this from the leader, passes it on to the appropriate process thereby granting resource to the picked process. This is the second o in the sequence. At this time, the remaining agents are still in c at

best. Once the resource is relinquished, the execution now moves higher up in the hierarchy (see Figure 3.8 thereby granting the resource to the next process in queue. By this time, the leader would have done a i, c and a o again. Another agent whose process is picked does a o also while still other agents are stuck at c. The event sequence is a arbitrary string of i's,c's and o's which implies that it is not in G. The execution correct because at most only one process has the resource. □

Both the type of templates provide the view of a centralized system to the user. The templates take advantage of the underlying architecture by dividing the work among agents and all this is transparent to the user. The algorithms in figures 3.4 and 3.9 run on a centralized system too.

Chapter 4

Fault-Tolerance

Any system or a model that is not designed to handle failures, hardware and software, is of less practical value even if it offers many other attractive features. It is not possible to control failures, they happen naturally. A system that continues to work despite failures is very attractive. Several types of failures are discussed in the literature, *timing, halting* and *byzantine.* We consider halting failures only, where a process stops executing without performing any incorrect actions. Halting failure is also referred to as *fail-stop* failure. The rationale behind this consideration is that this failure is very common. The proposed architecture, *Virtual Central Control,* tolerates process failures, hence it's fault-tolerant. There can be up to n-1 failures before the system degenerates to a centralized system. In this chapter we discuss the fault-tolerance feature of VCC.

4.1 Types of Failures

A computer system is a collection of components, both hardware and software, some of which may fail from time to time. The chances of failure exacerbates in distributed systems because they consist of a collection of computers and communication subsystems. Moreover, it is not possible to distinguish between a processor that is executing very slowly and one that has stopped[19, 21, 4]. Three types of failures are reported in the literature[4, 19, 12]:

- **Link Failure.** Failure of a communication link between the computer and the communication subsystem, for example, LAN.
- **Fail-stop.** A processor fails by halting. The other processors in the system can detect the failure.
- **Byzantine Failure.** A processor fails by exhibiting arbitrary behavior. For example, send contradictory messages, impersonate one another.

We consider fail-stop failures only. Also, as it is not possible to communicate in case of a link failure, we treat this type of failure as fail-stop.

4.2 Agent Failure

Every agent in VCC runs on a different physical computer. The agent process can stop working for many reasons, the computer may lose power, the operating system kernel may crash and so on. Even if the computer and the agent process are running, the communication link can severe. In all such cases, we say that the agent has failed as it is not possible to communicate with it. The agent-to-agent protocol given in Figure 3.2 is fault-tolerant. All the agents periodically poll other agents to make sure they are alive. The polling operation is done by sending a probe message to which the agent receiving the message will respond in a prespecified time-out period. If a response is not received, a *failure suspect* message is sent to the group. This message contains the information about the agent that is suspected to have failed. A second chance is given to the suspected agent by sending one more probe. If a response is received, the suspicion is dropped and the computation proceeds with this agent as part of the system. If not, this agent is dropped from the group.

If the agent that failed happens be the leader, one of the remaining agents will be chosen as the new leader. The way this happens is, all the active agents learn about the leader's death in finite time. The agents send a broadcast message calling for *leader election.* All the agents synchronize and elect a new leader. The computation proceeds as before but with a new leader.

Group membership is dynamic. Any agent, at its will, may chose to join or leave the group. Should an agent decide to take such an action, it is required to send a *join-group* or *leave-group* message. If an agent wishes to join the group, it will broadcast its intention to the group and only the leader will respond and the other agents ignore. The leader passes on the status of the group to the new member. All the operations discussed so far are transparent to the user.

4.3 Recovery

The agents are responsible for the execution of user programs. They are the critical components of the VCC and play a major role. It is not reasonable to have them execute user programs as an execution may not let them return to the service loop in time to perform normal activities. Separate user programs are spawned by agents.

Now, suppose that a user program has executed for some time before it fails or the agent crashes. A recovery of some sort is important as the other agents and user processes can proceed with the computation. We consider the user program and the associated agent as one entity. That is, if either of them fail, we say that both of them have failed and are no longer part of the system. Hereafter, we say "agents executing processes" even though there is a separate physical user process running.

After a failure, in some cases, VCC just repeats the step at which a failure occurred. This avoids the re-execution from the beginning which may be a timing consuming and an expensive operation as all the execution till that point has to be discarded. Starting all over involves scheduling of processes again and the messages that have to be retransmitted for synchronization. This is possible only in templates for synchronous mode. In asynchronous mode of operation there is no synchronization of any sort. The processes execute asynchronously so it is not possible to predict state of the system at any time.

4.3.1 Recovery in Synchronous Mode

If a program has been written using templates for synchronous mode of operation, the computation synchronizes at every *computation function* step. As discussed in 3.4.1, the semantics states that the partial results obtained after the execution of every computation function is sent to the leader. The leader waits till it receives partial results from all the agents. Thus the execution is synchronized. After a failure, the remaining active agents can roll back the execution to the last synchronization step and repeat the computation.

Let us consider the *maxima* problem again. Even though there is only one computation function in the algorithm (see Figure 3.4), it is sufficient to illustrate the point.

All the agents start executing the problem more or less at the same time. In the first step, they collect input from all the processes, in this case numbers. The next step, i.e., finding the local maxima, proceeds asynchronously as the agents can perform this step on their own. Now the partial results are sent to the leader. Let us now suppose that the leader fails. All the agents observe this failure and elect

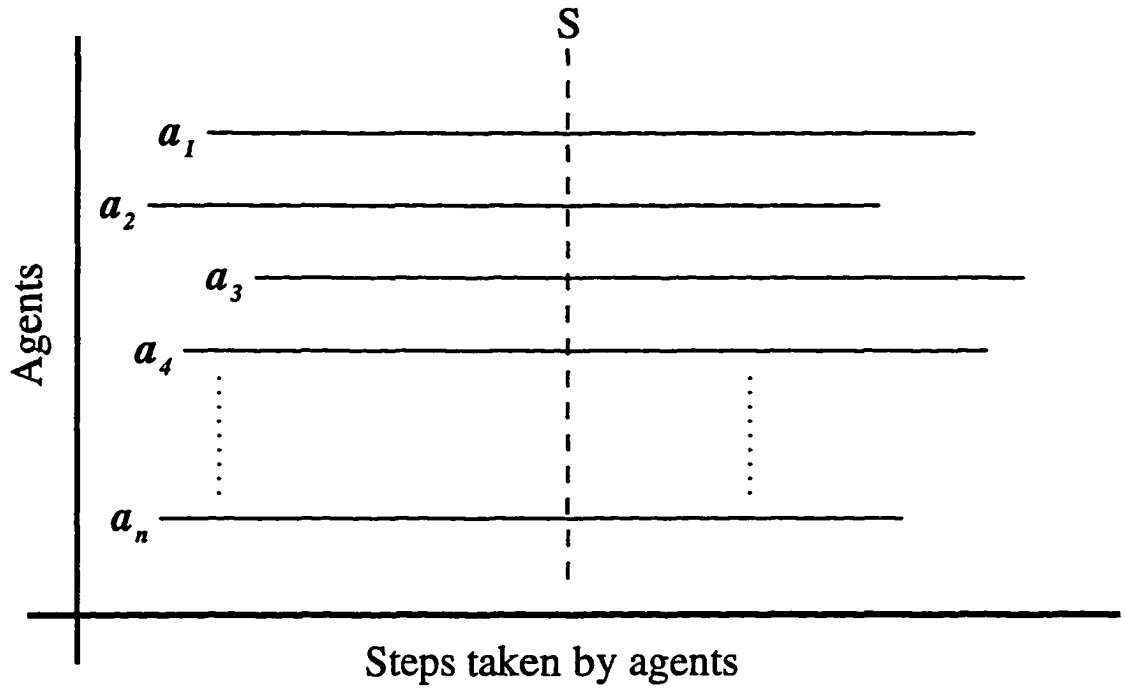

Figure 4.1: Execution Graph

a new leader. Once the leader is elected, the agents do not recompute the local maxima. All they do is retransmit the partial result to the new leader. Of course the local maxima the earlier leader had is no longer available and is not relevant to the computation anymore. The new leader proceeds with the computation to find the global maxima.

As a note, even though there are separate physical user processes executing, the execution of the computation function (finding maxima) is check pointed. And hence, after a *signal* from the leader, the user process restarts this step by resending the message to the new leader.

4.3.2 Recovery in Asynchronous Mode

The execution in this mode is unconstrained, the agents and processes execute asynchronously. The semantics are such that there are no synchronization points. For

1. Queue the requests for resource
2. Pick a request from the head of the queue
3. flag = 0
4. if picked_request = my_request
    a. assign the resource
    b. flag = 1
5. if (flag)
    a. wait for the process to relinquish the resource
    b. remove it from the queue
6. Goto step 1

Figure 4.2: One-shot Algorithm for Mutex

convenience, the execution hierarchy figure in Chapter 3 is given again in Figure 4.1. Once the computation starts, it is not possible to predict the state of the system which is important if the computation has to be rolled back. In other words, it is not possible to draw a line such as S in Figure 4.1, as the states of the agents and processes are arbitrary. One agent may be executing the input step while the other in computation and a third may have completed output and onto the next cycle of input. Thus it is not possible roll the computation few steps but to repeat it all over. The advantage of VCC over a centrally controlled system in this case is that the system doesn't come to a halt if the leader were to fail. A new leader will be elected and the computation proceeds. No doubt that discarding an execution and starting all over is expensive.

4.4 Mode Transformation

It is sometimes possible to express some computations, for example, mutex, as one-shot computation also. One such algorithm for mutex is given in Figure 4.2. Step 2 is a computation function. The agents pick a request and send it to the

leader. The leader in turn picks one of the several requests and broadcasts it to the agents. In step 3 (output function), the agents compare their request with the one returned by leader. The condition in step 4 will be satisfied only for one agent and it assigns the resource to the appropriate process. Only this agent waits in step 5. After the resource is relinquished, this process is removed from the queue. Other agents once again re-send their request as head of their queue has not been modified. If the resource has been assigned to one of the agents, the leader has to wait for this agent in step 2 before it picks a request again. This agent sends a request again only after the process to which the resource has been assigned relinquishes it (step 5). No more than one process is assigned the resource at any time.

Now that we saw a one-shot solution to the mutex problem, the next question is do all the computations have a one-shot solution ? The answer is affirmative.

Definition: A *Null event* is an event where no computation is performed. In other words, it is an empty event and is denoted by a bar on the event. $\overline{o}$, is a null output event, for instance. A null event can be inserted anywhere in the event sequence because it doesn't affect the computation.

We are interested only in input(i), computation(c) and output(o) events. A null input event means an agent has not performed an input action. Similarly, for computation and output events. If a snapshot of the event sequence has some events missing, it is safe to replace them with an appropriate null event.

Theorem 4.1 *All computations can be expressed as one-shot computations.*

Proof: It is sufficient to show that any sequence not in G can be transformed to one in G. Also, the transformed sequence should preserve the computation properties. Consider a sequence l in S such that $l \in S - G$.

$$l = i_2 i_1 c_2 i_3 c_1 o_2 i_j \ldots c_3 o_1 c_j i_m o_3 \ldots o_j c_m o_m$$

Here, we assume m agents generating *ico* sequences asynchronously. To transform this computation to a synchronous one, we have to introduce an artificial time line where we can force the execution to synchronize. We are interested in the first output event because all the event before it are i's and c's and so it is appropriate to synchronize at this point.

Now, consider a partial sequence of l,

$i_2 i_1 c_2 i_3 c_1 o_2 \ldots$

The partial sequence stops at the first output event, o_2. This sequence has to be transformed to one in G. We do so by adding the null event at appropriate places, as in,

$i_2 i_1 c_2 i_3 \overline{c_3} c_1 \overline{o_1}\ \overline{o_3} o_2 \ldots$

This sequence belongs to G. We can introduce the null events randomly as long as we preserve the *ico* sequence for any agent because the execution is asynchronous. This means that we can treat events that didn't occur at synchronization point as a null event. The only computation step in the sequence $i_2 c_2 o_2$. At a later point when the next o occurs, we reconsider the input and computation events that were not used at the previous synchronization step. This is the reason why agents retransmit their requests in the mutex algorithm above. Thus we transform all such partial sequences to get the final sequence that belongs to G. This implies all the computations can be expressed as *one-shot* computations. □

In this chapter we discussed the fault-tolerance feature of the VCC. In the synchronous mode of operation, the fault-tolerance is very effective. The computation has to be rolled only one step back. In case where it is not possible to express an algorithm in terms of a one-shot computation, asynchronous mode of operation

comes in handy. But in case of a failure, the computation has to be started all over. Also, we showed that every computation has a *one-shot solution.*

Chapter 5

Implementation

The very purpose of modeling is to provide an abstraction of an underlying theory or notion. Therefore, it is essential to test the model by building it, in this case implementing. This chapter gives the implementation details. A prototype has been implemented using Parallel Virtual Machine (PVM) [1] in 'C'.

5.1 Parallel Virtual Machine (PVM)

The PVM uses message-passing model to exploit distributed computing across a network of computers. The PVM software provides a unified framework within which distributed and parallel programs can be developed in an efficient and straightforward manner using existing hardware. PVM enables a collection of heterogenous computer systems to be viewed as a single virtual machine. It transparently handles all message routing, data conversion, and task scheduling across a network of different computer architectures.

The PVM computing model is simple yet very general, and accomodates a wide variety of application program structures. The programming interface is straightforward, thus letting simple program structures to be implemented in an intuitive manner. The user writes his application as a collection of cooperating *tasks*. Tasks access PVM resources through a library of standard interface routines that allow the

[1] is a collaborative effort of University of Tennessee and Oak Ridge National Laboratory

initiation and termination of tasks across the network as well as communication and synchronization between tasks. Communication constructs include those for sending and receiving data structures as well as high-level primitives such as broadcast, and barrier synchronization.

5.1.1 The PVM System

PVM is an integrated set of software tools and libraries that emulates a general-purpose, flexible, heterogeneous concurrent computing framework on interconnected computers of varied architecture. PVM supports,

- User-configured host pool
- Translucent access to hardware
- Process-based computation
- Explicit message-passing model
- Heterogeneity

5.2 The Toolkit

Some features of PVM are similar to that of VCC, especially, the unified view of a collection of computers over a network. Also, PVM provides high level routines for network computing like establishing a connection with a computer, sending a message, monitoring another machine on the network, etc. As VCC is based on message passing paradigm, many of the programming primitives supported by PVM was used to build to the prototype.

The toolkit has been implemented on the *Silicon Graphics* cluster in the Robotics Research Laboratory. Three workstations *power*, *wave* and *space* were used (see Figure 5.1) in the experimental run. An agent process runs on all the computers.

```
pvm> config
3 hosts, 1 data format
                    HOST     DTID     ARCH   SPEED
                   power    40000    SGI64    1000
                    wave    80000    SGI64    1000
                   space    c0000    SGI64    1000
```

Figure 5.1: Workstation Cluster

```
power 6% va
I am the leader 40003

pvm> ps
                    HOST      TID      PTID     PID   FLAG 0x COMMAND
                   power   (cons)         -    7836      4/c -
                   power    40002     c0001    7847    6/c,f pvmgs
                   power    40003         -    7848      4/c va
                   space    c0001         -    8740      4/c -
                   space    c0002     40003    8743    6/c,f va
                    wave    80001     40003    9426    6/c,f va
```

Figure 5.2: Configuration Showing Agents

Once the host pool is setup, agent processes are *spawned* on all the computers. All the agents form a group. During initiation, one agent process is started from the command line on one host, which in turn forks agents on the remaining hosts in the pool. The agents along with hosts is shown in Figure 5.2. There are three *vas* under the `COMMAND` column which shows three agents running.

Of the three agents, one of them is a leader. As shown in the Figure 5.2, the process with TID (task id) 40003 is the leader. This process has output the message "`I am the leader 40003`". If an agent other than the leader dies or chooses to withdraw from the system, the agent is dropped from the group. A new leader is elected only when the current leader dies.

5.3 Programs

This section explains how to write and run programs. Programs are written in 'C' and linked to PVM library.

5.3.1 Writing Programs

Let us consider the *maxima* problem. A skeleton program for the same is given in Figure 5.3. As described in Chapter 3, a program has to be expressed in terms of three functions. The *input, computation* and *output* functions must be tagged appropriately as shown in the first three lines of the Figure. A program can have as many functions, but all of them have to be tagged. The remainder of the program is a regular sequential 'C' program. Any program should include the files *"pvm.h"* and *"va.h"*.

```
INPUT  getinp()
COMP   maxima()
OUTPUT print()

#include <stdio.h>
#include <signal.h>
#include "pvm3.h"

getinp(int *arr,int *n)
{
   ....
   ....
}

maxima(int *iarr, int ic, int *oarr, int *oc )
{
   ....
   ....
}

print(int *oarr, int ocnt)
{
   ....
}

/* other functions */

main(int argc, int *argv[])
{

  /* declarations and other function calls */
  ..
  ..

  getinp(iarr, &icnt);
  maxima(iarr, icnt, oarr, &ocnt);
  print(oarr, ocnt);
  ..
}
```

Figure 5.3: Skeleton of a Sample Program

```
main(int argc, int *argv[])
{
        int iarr[size], oarr[size], icnt, ocnt;

        va_init(argv[0], atoi(argv[1]));

        iptempl(getinp, iarr, &icnt);
        comptempl(maxima, iarr, icnt, oarr, &ocnt);
        optempl(print, oarr, ocnt);
        .....
        .....

}
```

Figure 5.4: Modifying *function* Calls

A program will be written for one of the two types templates. It is the responsibility of the programmer to specify the appropriate switch at compile time so that the correct type of template object is linked.

Compilation

A program written this way is passed through a *preprocessor* that collects all the calls to the three types of functions and replaces them with calls to appropriate templates as shown in Figure 5.4, only the *main* part is shown. In the call, the function is passed as a parameter, which is a pointer to the function. Also, the template object module is linked at compile time. The *make* file provided does all the operations mentioned without the programmer having to know any details of the template module.

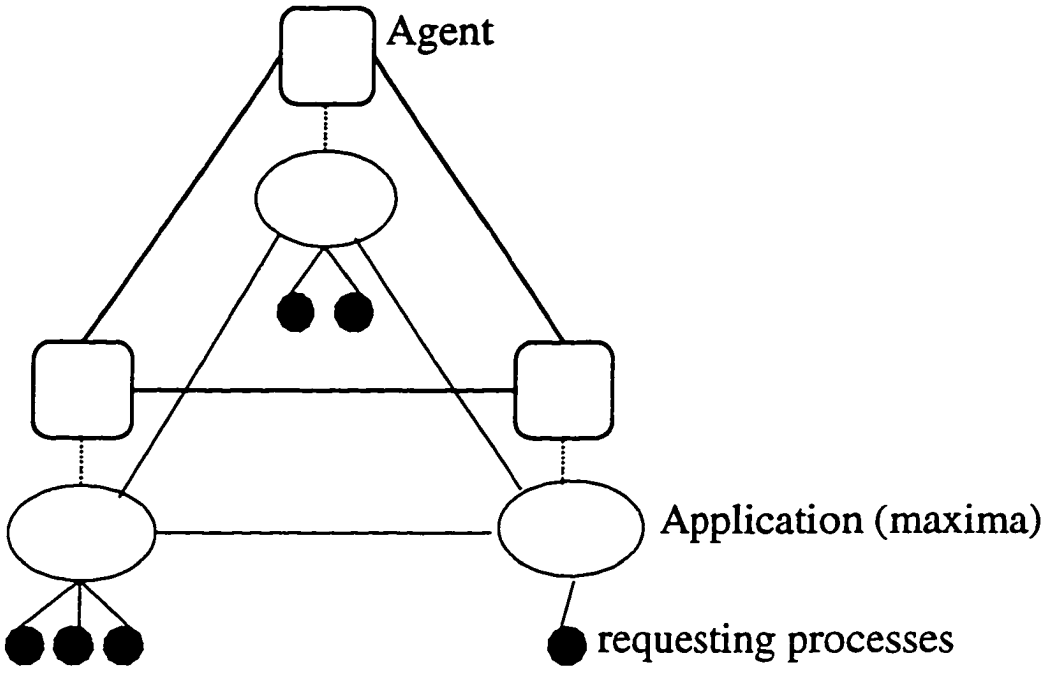

Figure 5.5: Agent and User Program

5.3.2 Running Programs

Once the program is compiled, a copy of the execution module must be made available to all the hosts in the host pool. If they happen to share a filesystem, one copy would suffice. Programs are executed through a separate program called *submit*. Programs can be executed on VCC from any of the hosts. For example, to run the maxima program, typing *submit maxima* on the console of one of the hosts would spawn the program on all the machines.

Submit broadcasts a message to the agents' group, the agents spawn the program on respective machines. As pointed out in 4.3, user programs are separate processes, see Figure 5.5. These processes form a group of their own, refered to as the program group. The program on the same host as the leader acts as *leader* to the program group. But the agents are in control of the respective programs in the sense that they are tightly coupled. In other words, they are treated as one entity. The failure

of either of them would cause both to be dropped from the system. All the actions mentioned above are transparent to the user.

In brief, to give a picture of how a program runs (see Figure 5.5), the agents and agents' group (indicated by thick lines) are setup at the system startup. Upon receiving a *submit*, the agents fork (dashed line) the user program, represented by an oval. The small black circles are *requesting processes* that send their requests to programs.

5.3.3 Printman

A simulation run typically requires some sort of output to examine the events or program state. In VCC, processes reside on different computers so it is not feasible to have them output the result on the computer they are running. Moreover, remotely forked processes do not have a terminal associated with them, so it is not possible to have *print* statements. Also, it is very important to view the system activities at one place. These reasons necessitate a tool for collecting the output from all the processes and displaying at one place - *printman* does this. Printman is spawned at system startup.

To redirect output to the printman, the following 'C' statements should be used.

```
char s[100];
sprintf(s, "The maxima is, %d", maxima);
print(s);
```

Where `print(s)` is macro defined in va.h that packs the string `s` and sends it to the printman. A printman output is shown in Figure 5.7. The number in brackets starting with a *t* are task ids.

```
pvm> ps
                HOST       TID      PTID      PID    FLAG 0x COMMAND
                wave     80001     40004     1234      6/c,f va
                wave    (cons)         -     1235        4/c -
               power     40002     c0002    15062      6/c,f pvmgs
               power     40004         -    15071        4/c va
               space     c0002         -     8320        4/c pm
               space     c0003     40004     8321      6/c,f va

pvm> kill 40004
```

Figure 5.6: System State after Startup

5.3.4 A Sample Run

The system status after startup is shown in Figure 5.6. Refer to Figure 5.7, in the fifth line, the task t40004 (one of the agents) has sent a message "I am the leader". The other two agents, with tids, t80001 and tc0003 have output the message "Leader is 40004". This means all the three agents are in consensus regarding who the leader is.

To test the fault-tolerance, the leader is killed deliberately. In the same Figure, the remaining two agents observe the failure and output the message "Leader killed". They confer immediately and elect one of them to be the new leader, the agent t80001. The outputs of these two agents reflect the fact.

Maxima problem

The output of a run of maxima program is shown in Figure 5.8. The output has been edited to fit in a page. The local maximas at the three agents are 76, 105 and 409. The program is run twice. There are no failures in the first run so all agents output 409, the global maxima. The message "User program submitted" marks the

```
space 3% pm
Printman (c0002) ready
[t40003]: ----------------------------------------
[t40004]: 0 - 40004
[t80001]: 1 - 80001
[t40004]: I am the leader
[t80001]: Leader is 40004
[tc0003]: 2 - c0003
[tc0003]: Leader is 40004
[t80001]: 1 - 80001
[tc0003]: 2 - c0003
[t80001]: Leader killed
[t80001]: after le
[tc0003]: Leader killed
[t80001]: I am the new leader 80001
[tc0003]: after le
[tc0003]: NOTME
[tc0003]: Leader is 80001
[t40005]: ----------------------------------------
```

Figure 5.7: Output of Printman

beginning of the execution and the programs terminates after the global maxima is found.

In the second run, after the program is submitted again, a delay is deliberately introduced to allow sometime to kill the agent with the highest local maxima to simulate failure. The start of the delay is signaled by the message "leader is sleeping". The leader happens to have the highest local maxima. Once the leader is killed, another leader is elected and the computation rolls back one step to find the new global maxima of 105. Note that the maxima before failure is not relevant to computation any more. Also, in the end of the figure, we see termination messages from two agents only.

```
wave 9% submit maxima
wave 10% submit maxima

space 5% pm
Printman (c0001) ready
[t40003]: 0 - 40003
[t80001]: 1 - 80001
[t40003]: I am the leader
[t80001]: Leader is 40003
[tc0002]: 2 - c0002
[tc0002]: Leader is 40003
[t40003]: User program submitted
[t80001]: User program submitted
[tc0002]: User program submitted
[t40005]: leader is sleeping
[t40005]: 409
[tc0004]: 409
[t80005]: 409
[t40003]: User program terminated
[tc0002]: User program terminated
[t80001]: User program terminated
[t40003]: User program submitted
[t80001]: User program submitted
[tc0002]: User program submitted
[t40006]: leader is sleeping
[t80001]: 1 - 80001
[tc0002]: 2 - c0002
[t80001]: Leader killed
[t80001]: after le 80001
[tc0002]: Leader killed
[tc0002]: after le c0002
[tc0002]: NOTME
[tc0002]: Leader is 80001
[t80001]: I am the new leader 80001
[t80007]: leader sleeping
[tc0005]: 105
[t80007]: 105
[t80001]: User program terminated
[tc0002]: User program terminated
```

Figure 5.8: Two runs of Maxima Program

This chapter discussed implementation details along with some examples. The goal of the model is to provide a simplified view that would ease the system development. The proposed model is implemented and the examples are given to verify the claim.

Chapter 6

Conclusions and Future Work

A distributed system is a conglomeration of components of various types and number. Naturally, it has the properties of any practical system: availability, reliability and scalability.

Various models have been proposed [2, 4, 5, 16, 20], each of which addresses a particular issue. In Chapter 2, we discuss other works that are closely related to ours. ISIS [5, 16, 17], a distributed programming toolkit that provides a view of synchrony, called *virtual synchrony*, in asynchronous systems. The system is built on three broadcast primitives, ABCAST, CBCAST, and GBCAST.

FrameWorks [20] is a system for the generation of distributed applications in a network of workstations. Modules that make up an application are written as sequential procedures. They are then encapsulated in one of the several types of templates with pre-defined characteristics. Templates describe the interaction of a module with other modules. This model does not provide methodologies that ease system development and is not fault-tolerant.

Distributed Application Framework[2] attempts to build large scale applications on top of network file systems. Applications are built from a collection of services offered by different servers on the network. This makes managing services as easy as managing files on a file system. A serious drawback is that if a particular service is not available the user has to build one. Also, the important features such as

synchronization, ordering etc. is left to the user which makes system development hard.

The *Virtual Central Control* architecture provides centralized view of a distributed system. This makes algorithm development easier. Algorithms are expressed in terms of three functions that have simple semantics. The requesting processes are given the illusion of a central coordinator as in a central system. A prototype is implemented that demonstrates all the features of VCC. The model along with protocols and algorithms are formalized to provide a theoretical base.

VCC offers a high degree of fault-tolerance. The system continues to operate till the last agent fails. An attractive feature of VCC is that in some cases the computation is rolled back by just one step in case of a failure. This avoids discarding the entire computation and starting all over which is computationally expensive.

6.1 Future Work

Extensions to this work can be made in several directions. First of them would be to have a stronger implementation by building process groups and other primitives that are more suitable to this architecture. This exercise possibly may provide a better insight towards the aspects that would be of practical importance.

The proposed architecture is more suitable for a LAN environment that supports computers of the order of hundreds. A system that supports computers over a wide area network is of greater importance. Extensions to *Agent-to-Agent* protocol to support multiple process groups to coexist and spawn *shadow*[1] agents to handle increased load would be a step towards building a large scale application.

[1]term borrowed from PVM literature

VCC currently supports three types of functions using which an algorithm is expressed. Quite a few variations were explored but this proved to be more effective in terms of simplicity and power. Even though it is proved that every computation can be expressed as a *one-shot* computation, a user may find it hard to so in some problems. Further, it is possible to "virtualize" the central agent by electing a different one after each "shot". Any refinements or modifications to the function structure that would automate or ease the conversion to a one-shot computation would be invaluable. It is possible breaking the functions in a different way might provide an elegant solution but remains to be explored. In VCC, asynchronous to synchronous transformation is expensive. Conditions that would weaken the synchrony is necessary.

Bibliography

[1] Michael D. Schroeder A State-of-the-Art Distributed System: Computing with BOB. *Distributed Systems*, Second Edition, Chap. 1.

[2] Rao, H. Distributed Application Framework for Large Scale Distributed Systems *IEEE 13th International Conference on Distributed Computing Systems*, 1993.

[3] Singhal, M. A Taxanomy of Distributed Mutual Exclusion *Journal of Parallel and Distributed Computing*, **18**, 1993, pp. 94-101.

[4] Mullender, S. *Distributed Systems*, Second Edition, Addison-Wesley, 1993.

[5] Birman, K. The process Group Approach to Reliable Distributed Computing *Technical Report 91-1216, revised Mar. 22, 1993*, Dept. of CS, Cornell University.

[6] Birman, K., Cooper, R. and Gleeson, B. Programming with Process Groups: Group and Multicast Semantics, *Technical Report 91-1185*, Dept. of CS, Cornell University.

[7] Ricciardi, A. and Birman, K. Using process groups to implement failure detection in asynchronous environments, *Technical Report 91-1188, Feb. 1991*, Dept. of CS, Cornell University.

[8] Turek, J. and Shasha, D., The Many Faces of Consensus in Distributed Systems, *IEEE Computer*, June 1992.

[9] Chang, J. and Maxemchuk, N.F., Reliable broadcast protocols, *ACM Trans. on Computer Systems*, **2**, 3, Aug. 1984, pp 251-273.

[10] Cheriton, D. R. and Zwaenepoel, W. Distributed process groups in the V kernel, *ACM Trans. on Computer Systems*, **3**, 2, May 1985, pp 77-107.

[11] Lynch, N., *Distributed Algorithms*, Lecture Notes, MIT.

[12] Lynch, N., *Distributed Algorithms*, Morgan Kaufmann Publishers, Inc., 1996.

[13] Lamport, L. Time, Clocks and the Ordering of Events in a Distributed System, *Comm. ACM*, **21**, 7, 1978, pp 558-565.

[14] Kaashoek, F. et al. An Efficient Reliable Broadcast Protocol, *ACM OPSYS Review*, **23(4)**, Oct. 89.

[15] Chen, M., Wu, K., and Yu, P. Efficient Decentralized Consensus Protocols in a Distributed Computing System *IEEE 12th International Conference on Distributed Computing Systems*, 1992.

[16] Birman, K., Schiper, A. and Stephenson, P. Lightweight causal and Atomic Group Multicast *ACM Trans. on Computer Systems*, **9**, 3, Aug. 1991, pp 272-314.

[17] Birman, K., Personal Communication.

[18] Birman, K., Joseph, T., Exploiting Virtual Synchrony in Distributed Systems *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, ACM SIGOPS, 1987, pp. 123-138.

[19] Coulouris, G., Dollimore, J. and Kindberg, T. *Distributed Systems, Concepts and Design*, Second Edition, Addison-Wesley, 1994.

[20] Singh, A., Schaeffer, J. and Green, M. A Template-Based Approach to the Generation of Distributed Applications Using a Network of Workstation *IEEE Transactions on Parallel and distributed systems* , vol.**2**, No. 1, Jan. 91, pp. 52-66.

[21] Sabel, L. and Marzullo, K. Election Vs. Consensus in Asynchronous Systems *Technical Report CS95-411, Feb. 1995*, University of California at San Diego.

[22] Geist, A., et. al. PVM:Parallel Virtual Machine, A Users' Guide and Tutorial for Networked Parallel Computing *The MIT Press, Cambridge, Massachusetts; London, England*, 1994.

Vita

Raghuram Yedatore was born on April 30, 1966, in Mysore, India. He completed his bachelor of engineering degree in Computer Science and Engineering from the University of Mysore, India, in 1987. After working for a short time, he came to the United States to continue his studies. He joined the Department of Computer Science at Louisiana State University in Fall, 1989. The day he met Vinayak Hegde, his fate was sealed to pursue his doctoral degree. He received his Master of Science and his doctor of philosophy degree in 1996. His areas of interest are Distibuted Systems, Operating Systems and Networking.

DOCTORAL EXAMINATION AND DISSERTATION REPORT

Candidate: Raghuram A. Yedatore

Major Field: Computer Science

Title of Dissertation: Virtual Central Control

Approved:

Major Professor and Chairman

Dean of the Graduate School

EXAMINING COMMITTEE:

Date of Examination:

9/12/96