UNIVERSITY OF ALBERTA

The Enterprise Executive

by

Pok Sze Wong

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of
Master of Science

Department of Computing Science

Edmonton, Alberta
Fall, 1992

Abstract

Enterprise is a graphical programming environment for designing, coding, debugging, testing, monitoring, profiling and executing programs in a distributed hardware environment. *Enterprise* code looks like familiar sequential code. The user attaches icons, called assets, to sequential code modules to specify the parallelism of the program. The system provides several kinds of assets, each representing a different high-level technique of parallelism. There is an analogy between assets in a program and those in an organization. The system automatically inserts the code to handle communication and synchronization into a module based on its attached asset kind. Also, the system runs the program and supports limited process migration and dynamic distribution of work, based on the demand and availability of resources.

Enterprise is an on going project involving many researchers. This thesis presents the work on the design and implementation of the textual interface, the *Enterprise* assets, the *executive*, and some practical applications to test the flexibility of the system. With the interface, the user can edit, compile and run an *Enterprise* program. Assets are implemented by adding communication and synchronization code characterizing their kind into user code. A C program preprocessor is used to turn a module call into a message sending the correct parameters. The *executive* manages the assets (and the machines they use) at run time. It runs assets on available machines and handles dynamic distribution of work, as well as concurrency issues such as fairness and termination. It also monitors the load on the environment and performs limited process migration when necessary. The *Enterprise* system has been used to convert several sequential programs to run on a distributed environment, as well as to develop new distributed programs. The experience indicates that the *Enterprise* model offers a flexible and easy to use approach for rapid construction of distributed applications.

Acknowledgements

I am indebted to the support and guidance of my supervisor Professor Jonathan Schaeffer. The readability and presentation of this thesis is largely due to his effort in reading the many drafts of the thesis.

I also thank Professor Duane Szafron, from whom I learn the object-oriented methodology, and the work has greatly benefitted from the support he provided. Thanks are due to Professor Werner Joerg, Xiaobo Li and Tony Marsland, my examination committee members, for their valuable comments and suggestions and for their time spent in reading this thesis.

I would like to thank the members of the *Enterprise* project group here, who provided helpful suggestion on this work. I wish to thank Enoch Chan who had to modify the compiler implementation many times, every change I made in the *Enterprise* executive. Ian Parsons is gratefully acknowledged for his bravery in using *Enterprise* to implement Baudet's chaotic Gauss-Seidel algorithm and his help in debugging the system. Carol Smith and Paul Lu also helped me in getting some applications to run on *Enterprise*.

Ajit Singh was the developer of *FrameWorks* which was the precursor to *Enterprise*. The design of *Enterprise* is much influenced by his experience. His work also made available to *Enterprise* a test suit of distributed programs. I would also like to thank him for his helpful comments on the design of the system.

Thanks also goes to the Department of Computing Science which provided me with technical and financial support.

Table of Contents

Chapter 1

Enterprise Parallel Program Model

1.1. Mission

This thesis is part of an ongoing research on the design and implementation of a distributed programming environment. The focus of this thesis is on the executive of the *Enterprise* environment. The term executive, although no longer used in the literature, refers to the computer's operating system. As Comer (1983) wrote:

> "Hidden in every computer system is the software that controls processing, manages resources, and communicates with external devices like disks and printers. Collectively, the programs that perform these chores are sometimes referred to as the executive, monitor, task manager, or kernel."

This term *executive* is chosen to be consistent with the anthropomorphic philosophy used in designing the *Enterprise* system and to distinguish the operating system in sequential computing.

The *Enterprise* system is built with four objectives in mind:

1) Provide a simple high-level mechanism for specifying parallelism that is independent of low-level synchronization and communication protocols,

2) Provide transparent access to heterogeneous computers, compilers, languages, networks and operating systems,

3) Support the parallelization of existing programs to take advantage of the investment in the existing software legacy, and

4) Be a versatile programming environment by eliminating the overhead that arises from switching between it and other programming tools.

The author's contribution to the project involves the following:

1) Participation in the overall design of *Enterprise* (E. Chan et al, 1991 and D. Szafron et al, 1992).

2) Developing a portion of the code generator to insert communication and synchronization code supporting a specified parallelization technique in *Enterprise*

into user modules. The entry procedure and module calls of the user code are turned into message exchanges (Chan, 1992) using a modified GNU C compiler.

3) Design and implementation of the *Enterprise executive*. The *executive* manages the *Enterprise* processes (and the machines they use) at run time. It runs processes on available machines and handles dynamic distribution of work, as well as concurrency issues such as fairness and termination.

4) Developing the textual interface of *Enterprise*. With the interface, a user can edit, compile and run an *Enterprise* program when a graphics terminal is not available.

The above work is reflected in this thesis. Chapter 2 discusses other similar work in the area of parallel and distributed programming environment. Chapter 3 and 4 demonstrate the *Enterprise* programming model. Chapter 5 outlines its overall architecture. Chapter 6 and 7 present the design and implementation of the *Enterprise executive*.

1.2. Motivation

During the last decade, there has been a steady shift from large centralized time sharing systems to networks of personal computers and workstations in commercial, academic, research and industrial environments. Several factors help to support this evolution.

Development in hardware technologies have lowered the cost of a processor. Also, advances in networking technology have made available high bandwidth communication access to facilities on the network. Regardless of the factors behind the change, the popularity of networked workstation computing environment opens up a new window of opportunity.

Parallel computing is considered as the most promising approach to high speed computing. Usually workstations are not used continuously, and there are often 'wasted cycles' as many of them are idle from time to time, especially during weekends and nights. The network of workstation combined delivers more power than most supercomputers ("the network is the supercomputer"[1]). Harnessing this computing power not only eliminates the need for costly larger computers and utilizes idle workstations, but has the additional advantage of providing the user with a uniform computing environment. The same workstation computer is used for all computing needs regardless of their computational complexity.

1.3. Problem

It is widely agreed that parallel and distributed software development involves much more effort and time than sequential programming. The design, implementation and testing of parallel

[1]A generalization by Carl Hamacher of Sun Microsystem's phrase "the network is the computer".

software is considerably more complicated than comparable sequential software. Efficient parallel algorithms have been developed for a vast group of problems, yet few algorithms find their way into practice. Obtaining a good parallel algorithm to a problem usually involves a small part of the total implementation cost. The major costs are spent in attacking issues that do not exist in the sequential environment. These include, for example, fault tolerance, synchronization, deadlock, communication, heterogeneous computers and operating systems, and the complexity of debugging and testing programs that may be non-deterministic because of concurrent execution.

Harnessing the computing power of a network of machines poses several challenges that do not have an analogue in the tightly-coupled multiprocessor environment.

1) The processors available to an application and their capabilities may fluctuate from one execution to another.

2) Communication costs are generally high in such an environment. This limits the kinds of parallelism that can be effectively implemented.

3) Users do not want to become experts in networking or low-level communication protocols to utilize the potential parallelism.

There are few systems that are aimed at providing shared processing power in a workstation environment, while taking into consideration the constraints of the environment and the user. There is a need for a tool for the rapid construction of parallel and distributed software, with a fault tolerance capability, and to harness the untapped potential of networked, idle workstations. This tool must bridge the perceived complexity gap between distributed and sequential software, without requiring the user to undergo extensive retraining.

1.4. Overview

Enterprise is a programming environment for designing, coding, debugging, testing, monitoring, profiling and executing programs in a distributed hardware environment. It represents the evolution of the *Frameworks* system (Singh, Schaeffer and Green, 1989a, 1989b, 1991; Singh, 1991). Programs written in the *Enterprise* environment look like familiar sequential C code since the parallelism is expressed graphically. The system inserts the code necessary to handle communication, synchronization and fault tolerance, allowing the rapid construction of correct distributed programs. This bridges the complexity gap between distributed and sequential software. *Enterprise* programs execute on a network of computers, using the idle cycles on the machines. The system supports load balancing, limited process migration, and dynamic distribution of work in environments with varying resource utilization. *Enterprise* offers an economical way to improve the productivity of programmers and the throughput of existing resources.

Enterprise has several features that makes it different from other parallel and distributed program development tools (see Chapter 2):

1) Users program in a sequential programming language (C) with new semantics defined for procedure/function calls that allows them to be executed in parallel. The semantic changes are simple and exhibit the powerful property that parallel code is indistinguishable from sequential code. *Enterprise* automatically inserts all the necessary communication and synchronization protocols into the user's code and frees the user from these details.

2) *Enterprise* can generate these protocols automatically because most large-grained parallel programs use only a small number of regular parallelization techniques (as will be illustrated in Chapter 2), such as pipelines and replication, etc. An *Enterprise* user selects a parallelization technique by manipulating icons using the graphical user interface. The user code for the parallel program is independent of the parallelization technique chosen. This enables applications to be adapted easily to a fluctuating number and type of available processors, usually without changing the user's code. It also provides an easy mechanism for experimentation and evaluation of how the different parallelization techniques fare on the user's specific application.

3) To make parallelism easy to express and understand, *Enterprise* uses an analogy between the structure of a parallel program and the structure of an organization. The new analogy replaces inconsistent analogies used in the past (pipelines, masters, slaves, etc.) by a uniform set of *assets* (contracts, departments, individuals, etc.). An organizational analogy was chosen because organizations are common and inherently parallel. This analogy provides the user with a different (but familiar) model for developing parallel programs. Although some researchers are against (Dijkstra, 1982) and some support (Booth, Schaeffer and Gentleman, 1982) the use of analogies in computing science, they should prove quite useful, as they have been in object-oriented programming.

4) Many of the parallelization methods supported by *Enterprise* can partition work dynamically in environments with varying resource availability. For instance, a *contract* supports partitioning of work to a varying number of identical subordinates. A contract utilizes as many idle machines as available to fulfill the task. Only a few

machines may be available during peak hours, but many more may be available in the evening to help complete the contract.

5) In many parallel computing environments, the user needs to draw communication graphs which usually involves drawing a diagram to connect nodes (processes) with arcs (communication paths). A similar diagram is created in *Enterprise*, but users are relieved from the tedium of drawing the details; they only have to edit the diagram by coercing and expanding nodes that represent assets. Coercion represents a high-level method to specify how a process (asset) interacts with its neighbors and corresponds to choosing a common parallelization technique. Expanding a node lets the user explore the hierarchical structuring of the application.

The user manipulates using the graphical interface and writes sequential code that is free of any parallel constructs. *Enterprise* recognizes where to insert the parallel code from the graph. It then compiles the routines, dynamically launches processes on machines and sets up the required connections. Processes run in the background, making use of the available idle machines and recognizing when machines become heavily loaded. Hence applications can profitably use machines that would otherwise be idle without interfering with the user community.

1.5. Scope and Organization of the Thesis

The emphasis of this thesis is on the *executive* of the system. The *executive* is implemented as part of the *Enterprise* system and as a result of this research. The thesis is organized as follows.

Chapter 2 gives a general discussion of issues of parallel programming environments, and reviews some existing systems.

Chapter 3 provides an introduction to the concept of the *Enterprise* model, including the new semantics of ordinary sequential procedure calls and the kinds of parallelism (assets) supported.

Chapter 4 illustrates programming on the *Enterprise* system using the textual interface.

Chapter 5 outlines the *Enterprise* architecture and its implementation.

Chapter 6 contains a description of the object-oriented design of the *Enterprise executive*.

Chapter 7 describes the implementation of the *Enterprise executive* using the ISIS package.

Chapter 8 summaries the thesis and provides a discussion of the ongoing research.

Chapter 2

Related Work

2.1. Introduction

Parallel and distributed computing are generally considered as the most viable approach to high speed computing. In the last decade, a wide range of multiprocessor architectures as well as the basic tools and techniques to utilize them have been developed. Also, there has been an increasing practice of using a loosely coupled multicomputer networks (e.g. a network of workstations), as if it were *a poor man's supercomputer*, to solve problems that used to be run on large centralized multiprocessor systems. In the past, the research in parallel computing was primarily aimed at achieving the highest possible performance levels. Most of the programming paradigms were modelled after the processor hardware. For instance, the programming models were usually categorized as shared memory or message passing (Bilardi, 1989; Sunderam and Geist, 1991). Such a restrictive view may be the fundamental reason why a large class of parallel algorithms have not made their way into actual implementation. Providing a programming paradigm that was easy to understand and use was not a major concern until lately.

In recent years, there has been an enormous increase in the number and quality of parallel programming environments described in the literature. Many of them have introduced new parallel programming models that are based on software perspectives. As mentioned in Chapter 1, this thesis is a continuation of these investigations. This chapter reviews the major techniques, tools and systems that are related to this work.

2.2. Scope Definition

Parallel computing is a diverse research area. Hundreds of parallel programming tools are being developed. It is important that the goals and scope of this review can be clearly defined and focused on topics related to the issues involved in this research.

Parallel processing can be viewed as part of the more general computing structure known as concurrent computing. In concurrent execution, there are more than a single thread of a program in execution. Concurrent execution is more commonly referred to as multiprogramming when several processes share a single processor at one time. The operating system multiplexes the processor among the concurrent processes. On the other hand, when the processes share a pool of processors simultaneously, it is called multiprocessing, or more often, parallel processing. Furthermore, if the underlying processors are connected by a local or wide area network, the computation is known as distributed processing.

The architectures of a parallel or distributed processing system can be categorized into two classes (Flynn, 1966). In single instruction multiple data (SIMD) architectures, every processor can be viewed as executing the same instruction simultaneously. In multiple instruction multiple data (MIMD) multiprocessor systems, every processor may execute different instructions. The MIMD architectures can be classified in terms of how the data is accessed, as shared memory or message passing. Furthermore, when the multicomputers are connected over a local or wide area network, the system is usually referred to as a distributed system.

As mentioned in the introduction, most parallel programming tools in the past supported a paradigm that is tightly coupled with the underlying hardware. This is comparable to the stage of assembler coding in the ancient days of sequential programming. With the advancement in parallel programming environments, paradigms that are independent of any hardware structures, focusing on the parallelization techniques themselves, are available.

In addition, a programming environment can be classified according to its features (Chang and Smith, 1990) such as the languages supported, automatic parallelization capabilities, fault tolerance, verification, testing, debugging and the hardware systems supported. In another related study, Bal, Steiner and Tanenbaum (1989) focus on the language issue in distributed systems. Most programming environments fall into a model of categorization introduced in Section 2.4. To limit this review to within the scope of this work, only programming environments that exhibit some conceptual programming models or paradigms for building parallel applications on large grain parallel architectures (using MIMD multicomputers and network of workstations) will be surveyed.

2.3. Review of Approaches to Parallelism

There are many ways to classify techniques for parallel computation (Bal, Steiner and Tanenbaum, 1989; Carriero and Gelernter, 1989; Chang and Smith, 1990; King, Chou and Ni, 1990). King, Chou and Ni (1990) characterize parallel computation from two perspectives: from the way the computation is partitioned and distributed, and from the way the computation is executed. This scheme appears to cover almost all the possible parallelization approaches. Based on this scheme, a new dimension is added to distinguish parallelization techniques that generate tasks to be dynamically distributed at run time and parallelization techniques that do not. The techniques to achieve parallelism are best summarized by a three dimensional scheme. Parallelization technique can be categorized from three perspectives:

1) From the partitioning and distribution of the computation,

2) From the execution style of the computation, and

3) From the mutability of the computation.

From the first perspective, computation can be characterized as data-parallel or function-parallel:

a) A function-parallel view has a program divided into subprograms of different functionality, which can be executed in parallel on different processors. Function-parallelism is sometimes referred to as parallelism by partition since the operation is partitioned into subtasks (Dennis and Van Horn, 1966). Function-parallelism is suitable for applications that can be programmed using many independent subroutines (animation, for example).

b) A data-parallel view has the data partitioned among the processors (Hillis and Steele, 1986). Processors execute the same program on different data subsets. Data-parallelism is suitable for applications which perform the same set of operations repeatedly and independently on a large set of data. Data-parallelism can also be referred to as parallelism by replication, since the operation is replicated on different sets of data. Programs with nested loops to handle static and regular data structures are suitable for data-parallel computation (image processing and ray tracing, for example).

From the second perspective, execution, a parallel computation can be characterized as either concurrent or pipelined (Hwang and Briggs, 1984):

a) Concurrency exploits spatial parallelism by utilizing different processors working simultaneously on independent tasks which may be data-parallel or function-parallel.

b) Pipelining exploits temporal parallelism. Each processor works like a filter on its input data and passes output data to the succeeding processors. Data flows through the pipeline as they are processed stage by stage. A datum, once entering the system, will be used or modified repeatedly along the pipeline.

From the third perspective, mutability, a parallel computation can be characterized as either static or dynamic (Suhler, Biswas, Korner and Browne, 1990). Generally there is more logical parallelism in a parallel algorithm than physical parallelism in a multiprocessor. At run time, there may be more ready tasks than available processors.

a) In static parallelism, the distribution of tasks is performed statically at load time. The static parallelization approach is used when the computation structure is regular, mapping nicely to the physical computation structure. The approach is also used when the computational focus does not change with time.

b) In dynamic parallelism, the distribution of task varies with the demand of the computation. The programmer is relieved from predicting the resource utilization characteristics of the program. When the program exhibits a computation structure that is irregular, or when the computation focus changes with time (as in combinatorial search problems), it is impossible to achieve a good mapping between tasks and processors (and the less than optimal load balancing degrades performance).

By combining the three perspectives, different styles of parallel computation can be identified.

2.3.1. Static and Dynamic Concurrent Function-Parallel Computations

In these approaches, different processors perform different functions simultaneously. The strategy is to break the original problem down into several small independent subproblems. Each processor executes a different function on the same data. The parallelism is static in that each subtask involves approximately the same amount of work. For example, in a particle-in-a-cell application, different boundary constraint tests can be applied in parallel to the same datum (Singh, Weber and Gupta, 1991). The parallelism is dynamic when the amount of work involved in solving the subproblem varies in the execution. Some simulation applications can take advantage of this kind of parallelism (Myrias, 1990).

2.3.2. Static and Dynamic Concurrent Data-Parallel Computations

Processors perform the same operation simultaneously but on different data sets in the concurrent data-parallel approaches. These approaches are also commonly known as domain decomposition (Fox et al. 1988). Some problems exhibit a static computation structure and the amount of input data fed to an operation is constant, as in problems like parallel π computation. On the other hand, in some problem, the amount of data required to be processed changes with time, as in parallel sorting applications.

2.3.3. Static and Dynamic Pipelined Function-Parallel Computations

In the pipeline different stages perform different functions. When data flows through the stages, they are modified along the way. Traditionally, in terms of computer architecture, the pipeline hardware has all the operations taking the equal amount of CPU time of one tick. A branch and bound algorithm exhibits pipeline parallelism as well as concurrent parallelism (Schaeffer, 1989). Each node in the search tree would take roughly the same amount of time. The *animation* application discussed in this report illustrates an application suitable for a pipeline (Foley van Dam, Feiner and Hughes, 1990). This application also shows that during execution, some of the operations in the pipeline may require more computation time than others. Preferably, more processors should be allocated dynamically to meet the demand.

2.3.4. Static and Dynamic Pipelined Data-Parallel Computations

The data-parallel approach has a number of processors performing identical operations on individual data elements. To carry out correct executions, different data streams have to flow along different directions to bring appropriate data to the required processors at the right moment. The operations are synchronized. Since the communication overhead associated with medium and large-grained multicomputers are considerable, and a global clock is required to synchronize the operation among the processors, this approach of parallelism is uncommon in MIMD and distributed parallel programming environments and is most often seen in fine-grained granularity applications (such as systolic systems (Kung, 1979)). Carriero and Gelernter (1989) do not even consider this approach. However, research has been underway to explore its feasibility (King, Chou and Ni, 1990) and some programming environments support it (Segall and Rudolph, 1985).

2.4. Models of Computation

It is widely believed that the abstract computation model provided is the heart of a programming environment, whether it is parallel or sequential. In the world of sequential programming, we usually program at a conceptual level using a high-level language and seldom concern ourselves with the physical architecture of the machine. In the world of parallel programming, however, the idea of a conceptual machine is not as common. The parallelism in a program is usually expressed in an architecturally specific way, forcing the programmer to consider issues such as communication and synchronization at a low level. Recently, the idea of using an abstract model that is most natural to the problem to describe parallel computation has been addressed by many researchers. According to them, in general, the formulation of parallel computation can involve the following steps (Carriero and Gelernter, 1989; Browne, 1985, 1986):

1) Choose the model of computation that is most natural for the problem.

2) Write a program using the method that is most natural for that abstract machine.

3) If the resulting program is not acceptably efficient, transform it methodically into a more efficient version by switching from a more-natural method to a more-efficient one.

Programming at the conceptual level has many advantages. It increases the portability and reusability of the program. It also allows the parallelism in a program to be expressed in a natural way, implying that the programmer need only be knowledgeable in his application, not in parallel programming. Therefore the strength of a parallel programming environment is largely affected by the computation model it provides.

This section gives an overview of recent developments in parallel programming environments, classifying them by their conceptual computation model and comparing them on their strengths and weaknesses in expressing parallelism. The survey is not exhaustive but rather gives a representative sample of the current researches. The *Linda* language (Gelernter and Carriero, 1985; Gelernter, 1989; Carriero and Gelernter, 1988, 1989) uses the concept of a *tuple space* for communication between concurrent processes. Processes use atomic operations to read, read and delete, or add a tuple to the tuple space. This model is powerful but it does not provide a high-level parallel conceptual machine to describe a parallel program. This review therefore does not include *Linda*. Another relevant parallel programming model is the *Chare* kernel (Fenton, et al, 1991). A *chare* is a module to be executed in parallel. It can have multiple entries in which enclosed program segments and new chares can be created (even recursively) to execute the entries. The portability of applications is a tremendous asset, however the onus is still on the programmer to use the supplied programming language enhancements and explicitly code all the parallelism. As with *Linda*, *Chare* kernel is not a high-level design tool for parallel programming and is not covered in this survey.

2.5. Classification Scheme

The models of parallel computation can be described as a directed graph whose nodes represent units of computation, and whose arcs represent the data flow, control flow or messages between nodes (Browne, 1985). The models can be classified into three categories according to the structural focus employed to incorporate the parallelism: data oriented, task oriented and template attachment as Figure 2.1 (modified from Pancake and Utter, 1989) shows.

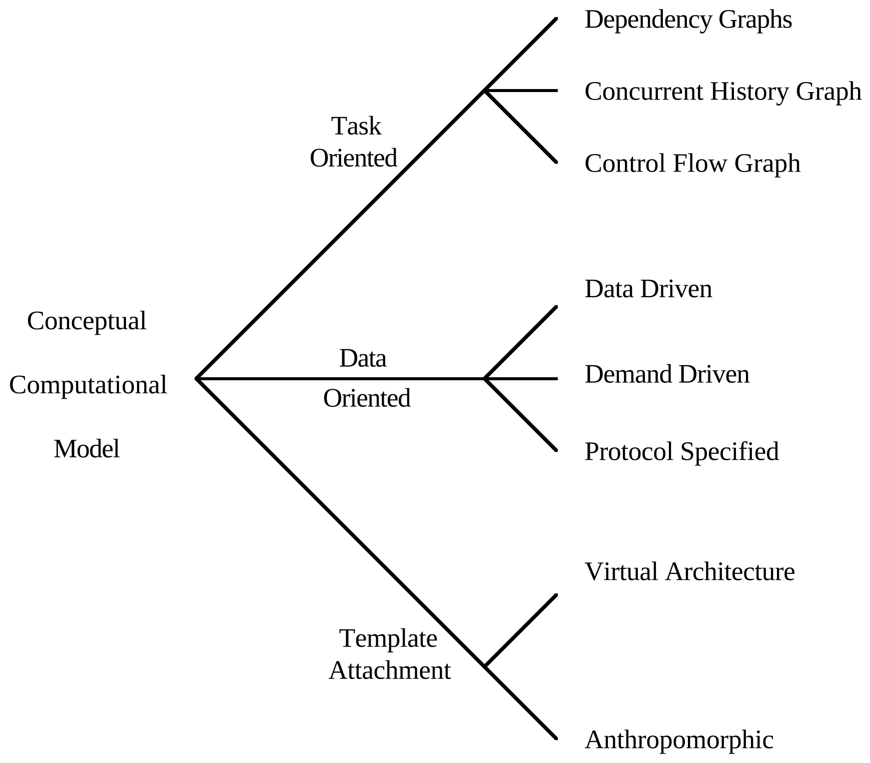


Figure 2.1. A Classification Scheme for Parallel Programming Environment Models

2.6. Task Oriented Models

In these models, parallelization is achieved by partitioning the tasks to be performed into a collection of serial processes, each processing a unique thread of control. Models in this category can be represented by a directed graph whose nodes represent a procedure call or a synchronization point, and whose arcs represent an operation that advances the program from one state to another. These graphical environments usually support dependency analysis of programs which are written in Fortran and run on machines with a small granularity (for example, PAT: Smith, Appelbe and Stirewalt, 1990; E/SP: Sridharan, et al., 1989; CAPER: Sugla, Edmark and Robinson, 1989).

2.6.1. CAPER

CAPER (Sugla, Edmark and Robinson, 1989) is an application programming environment for message passing multiprocessors. A program is specified by drawing a graph of nodes and lines. A node contains routines implementing algorithm or transforming data between communicating algorithms (which distribute and restructure distributed data). A line represents the transfer of data between two routines. The programmer can program a parallel application by connecting a standard working set of parallel programs together and specifying the necessary data transformation routines. With the graphical interface, this would involve drawing lines between icons representing the program pieces. When the problem involves the use of a new algorithm, the programmer can construct new pieces of code rather than using the working set, and connect the pieces together. Each node is also assigned a number which specifies how many processors are used to run the encapsulated routine. The graph is hierarchical and nodes can be grouped together or expanded.

2.6.2. E/SP

E/SP (Harrison, 1990; Sridharan, et al., 1989, 1990; Browne, Sridharan et al., 1990) is an environment for the parallel structuring of Fortran programs using a hierarchical dependency graph. The highest level graph of a program consists of a computation node, an entry node, and an exit node. A node of the most resolved level contains single statements. The next level of resolution is the basic block or the subprogram, and the highest level is typically a segment of a call tree. The Fortran language has been extended to include *fork* and *join* statements for expressing parallelism. The programmer collapses or expands a node to decide the granularity of a computation unit and the realization of the parallelism is done automatically by the tool. When dependencies must be removed to achieve parallelism, the programmer is informed of the dependency type and prompted to change the portion of the program involved. The drawback is that only the divide-and-conquer paradigm of parallel processing can be realized.

2.6.3. PAT

Parallelizing Assistant Tool, PAT (Harrison, 1990; Smith and Appelbe, 1988; Smith, Appelbe and Stirewalt, 1990), contains a parallelizer which examines Fortran source code and suggests parallelization modifications, a static analyzer that simulates the execution of the source program and locates anomalies caused by the interaction of tasks, and a debugger. The program analysis is built using a control flow graph (CFG) of the program. Each node in the CFG represents either a basic block of the program, a branch or a merging of the program flows. Subroutine calls in the CFG are expanded inline for simplicity (each call is expanded individually in context). The dependence information is extracted by tracing paths through the program, using reference lists to construct a global dependence graph. Browsing of dependencies is provided through an interactive graphical interface at the statement or variable level.

2.7. Data Oriented Models

For data oriented models, parallelism occurs as the simultaneous execution of an operation on multiple data elements. Whenever the size of the data warrants parallelization, activities are replicated for execution across data subsets. Program execution is viewed as a single thread of control which temporally diverges into parallel action sequences that later converge. Models in this category can be represented by a directed graph whose nodes represent units of computation and whose arcs represent flow of data (Babb, 1984). Models can be further classified as being demand-driven, data-driven or user-specified.

The sequence of operations in the demand-driven model of computation is that the sink node requests data from the source node and the source node responds. The sink node is the terminating node producing the final output. The source node is the initial node in the program which is also responsible for getting the input data. From the sink node, each node first sends a *demand for data* request to its parent in the dependence graph. The demand is then propagated towards the input of the graph. From the source node, the result is then propagated back to the sink node. A data-driven methodology employs a communication protocol where source nodes send results to sink nodes and perhaps await acknowledgement of result arrival or use a time-out mechanism. In both cases, the functions are embedded in the nodes and data objects are carried on the arcs. The number of messages in the demand-driven model is double that of the data-driven model and the size of the demand messages are small, resulting in a poor utilization of bandwidth. Some models (for example, CODE, (Sobek, Azam and Browne, 1988)) integrate demand-driven and data-driven control by allowing some arcs to be data-driven and others to be demand-driven. Most of the existing parallel programming environments specify parallelism based on the large grain dataflow model developed by Babb (1984).

2.7.1. CODE

Computation Oriented Display Environment, CODE (Sobek, Azam and Browne, 1988; Browne, Werth and Lee, 1989), uses a large grain dataflow model. Computations at each node can be specified as data- or demand-driven. The programmer describes the computation by drawing a graph of nodes and dependencies using icons chosen from a palette. The graph can be hierarchically defined; a node may contain a subgraph of nodes, a filter, or a subprogram. A node that contains a program is called a schedulable unit of computation (SUC). Code associated with a SUC is entered using a text editor. SUC's communicate via dependencies. A filter node defines the transmission of some subset of data from its input dependencies to some subset of its output dependencies.

2.7.2. DGL

Direct Graph Language, DGL (Jagannathan et al., 1989), is a parallel programming environment based on a demand-driven, large grain dataflow model. Every node is associated with a function module. Each incoming edge is associated with a parameter taken by the function. Each outgoing edge is associated with a use of the result of the function. The programmer draws a dataflow graph using a graphical tool to represent the dataflow of the program. The function can return only a single result and have no side effects. This is the major limitation of the system, since many applications, such as graphics programs, modify their data (for example, frame buffer or the terminal screen) through side effects. Another drawback is that it supports only a single level resolution of parallelism.

2.7.3. HeNCE

The programming environment for a heterogeneous network of parallel machines, HeNCE, (Beguelin et al., 1990, 1992) supports the creation, compilation, execution, debugging, and analysis of parallel programs for a heterogeneous group of computers. The programmer specifies the parallelism of a computation by drawing a directed acyclic graph where the nodes in the graph represent subroutines and the arcs represent data dependencies. HeNCE will automatically insert code to handle communication into these procedures and run them on machines in the network. In addition to the standard dataflow paradigm, HeNCE also supports dynamically spawned subgraphs, pipelining, loops and conditions. In general, HeNCE is similar to TDFL.

2.7.4. LGDF

Large Grain Dataflow, LGDF (DiNucci and Babb, 1989), takes the form of a directed graph consisting of processes (represented as circles) and datapaths (represented as vertical lines or rectangles) selectively connected pair-wise by arcs. A process is a sequential program written in a high-level language such as C or Fortran. Each datapath contains a (possibly zero length) data structure specified by the programmer. For a process to read and/or write to this structure, the process and datapath must be connected with an arc having read and/or write permission. These permissions are represented in the graph as arrowheads on the arc illustrating the allowed direction(s) of dataflow: towards the process (for read permission) and/or towards the datapath (for write permission). At any moment, a datapath can be in the state of being read from, written to or neither. A process may start execution (fire) when it can read from all of its connected arcs. Also, by applying the *reserve* and *grant* commands to a datapath, a process can exclude or permit other processes from accessing the datapath. It supports only single level resolution of parallelism.

2.7.5. Paralex

Paralex (Babao$\overset{u}{g}$lu, Alvisi, Amoroso and Davoli, 1991) is a parallel programming environment based on a data-driven, large grain dataflow model. The computation satisfies the functional paradigm. Data communication is specified by drawing links to connect the nodes. There are two kinds of nodes. At a computation node, execution begins when data arrives at every link incident at a node. At a filter node, data values are extracted on a per destination basis before they are transmitted to the next node to avoid transmitting unnecessary data. The user selects a node type from a menu and the tool puts the node on the screen. The drawing of links involves only clicking the mouse on the source and destination nodes. This is a major advancement over most other tools since the user is free from the tedium of most of the graphical layout task. The drawback of this environment is that it exhibits only a single level resolution of parallelism.

2.7.6. PPSE

Parallel Programming Support Environment, PPSE (Lewis and Rudd, 1990), is an integrated set of tools for the design and construction of parallel software. The PPSE Parallax graphical design editor supports a hierarchical graphical description language called ELGDF (Extended Large Grain Dataflow) to describe data and control dependencies between tasks. It offers graphical constructs such as pipes, loops, repeated nodes, and special dataflow arcs to indicate mutually exclusive access to data. The user attaches to each sequential routine one of the graphical nodes and connects the nodes together to represent the dataflow between different routines. It supports hierarchical resolution of parallelism.

2.7.7. TDFL

Task-Level Dataflow Language, TDFL (Suhler, Biswas, Korner and Browne, 1990; Suhler, 1989; Suhler, Biswas and Korner, 1987) evolves from CODE. TDFL introduces the idea of mutable computation graphs to support the mutable execution of recursive functions. If the data input to a function is above a threshold size, then additional processes are created to help process the data. The creation of nodes continues until the data size is within the threshold. However, data size may not be the best criterion to determine the degree of mutability, since processor usage is not necessarily proportional to data size. Since it is difficult to describe loop and do-all parallelism using the dataflow model, TDFL introduces new constructs to support parallelism not addressed in the formal dataflow models: *Loop*, *DoAll*, *Case* and *EndCase* nodes, and self-loop arcs. TDFL is a demand-driven dataflow system and therefore exhibits implicit parallelism. As in other dataflow parallel tools, the programmer uses a graphical tool to generate a program graph, and text is used to specify the code.

2.8. Template Attachment Models

Yet another class of approaches provides a set of predefined parallel objects which defines the communication and execution behavior of the enclosed module. Among the earlier efforts, the Parallel-programming and Instrumentation Environment (PIE) made available to the programmer pre-coded templates of parallel structures such as master-slave, pipeline and systolic multidimensional pipeline (Segall and Rudolph, 1985). In these models, a template provides an implementation of the communication and synchronization structures and the programmer needs to provide only the sequential code. Some environments model the computation entities after physical architecture characteristics and provide architectural templates like systolic arrays and master/slave relationships (for example, PIE: Segall and Rudolph, 1985; PAL: Xu and Hwang, 1989). Some environments support the techniques of parallelism directly through the templates (for example, *FrameWorks*: Singh, Schaeffer and Green, 1989a, 1989b, 1991; Singh, 1991). They focus on problem solving at a higher level using some anthropomorphic analogies.

2.8.1. PAL

Parallel Language, PAL (Xu and Hwang, 1989), is a procedural parallel language introducing the language construct *molecule*. A *molecule* is a set of program objects that have some common properties. A *molecule* type characterizes a particular computation mode (SIMD, sequential, pipelining, array processing, dataflow, multiprocessing, etc.). The user associates a sequential procedure with a *molecule* type and the procedure then executes in the specified mode (SIMD, sequential, etc). A new *molecule* type can be created by specifying the operations characterizing the type (for example, a stack machine can be defined through the push and pop operations).

2.8.2. PIE

In the Parallel-programming and Instrumentation Environment, PIE (Segall and Rudolph, 1985; Vrsalovic et al., 1988; Harrison, 1990), parallelism is realized by *join* and *detach* statements. Global data, and operations on that data, are encapsulated in *frames*. Frames are shared among specific tasks and/or C functions and shared abstracted data types can be constructed using frames. The most important idea in PIE is the implementation template construct, which defines and controls parallel computation activities and data. The pre-coded structure of an implementation (in terms of control, communication, synchronization, and data partition) is made available to the programmer in the form of a modifiable template. The template provides the send/receive operation and the programmer has to provide only the sequential code. Implementations can be nested with one implementation being a part of the other. The following implementations are available: master-slave, recursive master-slave (same as master-slave except that the slave could become recursively master for another set of activities), heap implementation (in which work is distributed through data structures like queues or heaps), pipeline and systolic multidimensional pipeline.

2.8.3. FrameWorks

In *FrameWorks* (Singh, Schaeffer and Green, 1989a, 1989b, 1991; Singh, 1991), an application is viewed as a graph with nodes being communicating processes. Each node contains a sequential module or procedure. Communication and synchronization are specified by the properties of the node through up to 3 types of attribute bindings. The input template specifies the incoming message to a node, and can be one of the following: initial, which accepts no input from other nodes; in_pipeline, which specifies the node as a part of a pipeline; and assimilator, which states that the node merges the results of several nodes. The output template specifies the outgoing message from a node, and can be one of the following: out_pipeline, which specifies the output of the node to flow in a pipeline fashion to its connecting nodes; manager template, which specifies that a fixed number of multiple workers will be used to execute the called procedure; and terminal template, which specifies that the application terminates there. The body template defines the execution mode of the node: an executive template causes the process to have its input, output and error streams directed to the terminal; and a contractor specifies that multiple workers are assigned dynamically at run-time to execute the specified node. The run-time dynamic task decomposition is supported by the language constructs *split* and *merge*, which are similar to *fork* and *join* in task-oriented models.

FrameWorks introduces the mechanism in which processes are replicated dynamically according to availability of idle processors. However, its method to specify the parallelism may be less intuitive to many programmers. Programmers have to go through a mental process to transform parallelization techniques and express them in terms of input and output communication.

2.9. Conclusions and Comparative Remarks

The data dependence graph contains redundant information for the parallelization process. Many dependencies are generated by scalar variable references, which offer little possibility of parallelism. Also, it is rare that every statement is involved in a compute-intensive segment. Storing the data dependency information for a region that is not compute-intensive is wasteful, since parallelization of it achieves little. A model that supports hierarchies of abstraction permits the computation to be realized with increased resolution and limits the size of the graph to a manageable complexity. The highest level graph consists of a single node which contains the entire program. The node of the most resolved level contains single statements. All three categories of models can support hierarchical abstractions of the computation graph that describes the parallelism of the program. In data oriented and task oriented models, the graphs are strictly hierarchical since expanding a node gives a subgraph with similar properties as the original node. In template attachment models, graphs at different levels may exhibit different properties.

Nodes in the template attachment models, by definition, may exhibit different properties. In general, it can express any parallelism desired by designing new templates. The other two models, task and data oriented, are more limited because they usually support a single node type - the computation unit. Both these models work best for expressing concurrent function-parallel computation. It is possible to express concurrent data-parallel parallelism only when the data size and number of processors are static. Besides, to express a simple concurrent data-parallel computation in which each processor takes an equal share of the tasks and works in parallel, the programmer has to partition the data and specify the routing of data for each node. Also, most models do not support recursive calls and dynamic concurrent data-parallel computation. Finally, it is impossible to express pipeline computations in these models. Therefore, some environments introduce special nodes to express the above parallelisms. In TDFL, the *DoAll* node is added to the dataflow model, which takes an incoming array data token and outputs an array data token. The size of the array determines the number of times the function is invoked. Each invocation writes to a different element of the array. Also, the *Self-Loop Arc* is added to support state retention.

In some cases, the dataflow diagram can be automatically translated from the dependency specification in the file (*makefiles* are a good example). Non-procedural languages, such as Lisp or Prolog, are better described by a data-oriented model. However, usually, dataflow diagrams are not readily available.

Because of the extensive research in the area of automatic parallel compilers, the program call structure and control flow information can be automatically generated in most cases. The programmer usually needs to deal only with the unresolved dependencies. However, the parallelism obtained by an automatic parallelizer may not be optimal since the analysis is based on static information only.

When the dependency information is not readily available, it is time consuming and error prone for the programmer to draw a large number of nodes and edges when the data flow or call structure are complicated. Moreover, data dependence, program control flow and program states are all low-level information and can often be non-intuitive to the programmer. A tool that interacts with a programmer only in terms of the above information can easily overwhelm the programmer with a large amount of low-level information. Also, these graphs can be dense structures. For large programs, this may pose a manageability problem to the programmer.

If the concern is to design a parallel program with the programmer in control, the best choice would be a parallel programming environment that uses the template attachment model. If mutability and implicit parallelism are the concern, then the data oriented environment is the choice. A task oriented environment is best used as a tool to let the programmer view and modify the program after automatic parallel structuring has been performed.

Chapter 3

The Enterprise Model

3.1. Introduction

In *Enterprise*, there is generally no difference between the organization of a parallel or distributed program and that of a sequential program. The structure of the program is independent of whether it is designated for executing sequentially or distributed. An *Enterprise* program is a collection of modules with each having a single *entry procedure* that can be called by other modules and a (possibly empty) collection of *internal procedures* that are known only within that module. Sharing of variables among modules are not allowed. This is comparable to programming with abstract data types, which provide well-defined means for manipulating data structures while hiding all the underlying implementation details from the user.

The code is executed sequentially within any module. A sequential program is therefore simply a single module *Enterprise* application with the module's entry procedure being the main program. *Enterprise* introduces parallelism by allowing the user to specify the way in which the modules interact. Module interaction is determined by two factors: the *role* of a module and the *call* to a module. The role of a module specifies which one of a fixed set of parallelization techniques (*asset* kinds) the module will use in execution. The call to a module defines the identity of the called module, the information passed and the information returned. The role of a module is expressed graphically while the call is specified in the code.

3.2. Module Calls

Procedures interact using procedure calls in a sequential program. The calling procedure, say A, has a procedure call with a list of arguments to a procedure, say B. Procedure A is suspended after the call is made and procedure B is started. Procedure B can use the information passed as arguments. B sends the results to procedure A when B has completed execution, through side-effects to the arguments and/or through the return value if the procedure is in fact a function.

It is useful to distinguish calls that return a result from those that do not. The former are abbreviated as *f-calls* (function calls) and the latter are abbreviated as *p-calls* (procedure calls). *Enterprise* module calls are similar to sequential ones except in two ways:

1) Arguments cannot be pointers, nor can they contain any pointers. Therefore, in the C-language version of *Enterprise*, module calls cannot return values via side-effects

since C uses call-by-value as its parameter passing mechanism. Other than that, there is no syntactic difference. A procedure or function may be called with an arbitrary number of parameters. Also, there is no limitation on the data type which a function should return.

2) When a module A calls another module B, A is not suspended but continues to execute. The p-call in the statement:

```
B(data);
/* some other code */
```

is non-blocking, so that A continues to execute concurrently with B. Of course in this case, B does not return a result to A.

However, if the call to module B was an f-call, then module A would block when it tried to use the function result, if module B had not yet completed execution. Consider the following example:

```
result = B(data);
/* some other code */
value = result + 1;
```

When this code is executed, the calling module A, only blocks when the statement "*value = result + 1;*" is executed and only if module B has not yet returned the value of *result*. This concept is similar to the work on futures in object-oriented programming (Chatterjee, 1989).

The syntactic similarity between procedure calls and module calls makes it easier to transform sequential programs to parallel ones and makes it trivial to change parallelization techniques using the graphical user interface. Usually no modification to the user's code is required.

3.3. Module Roles and Assets

The role of a module is independent of its call and depends only on the parallelization technique chosen. There are a fixed number of predefined roles corresponding to *asset* kinds.

To help describe module roles, an analogy between *Enterprise* programs and the structure of an organization is used. An organization usually has various assets available to perform its tasks. For instance, a large task could be split into subtasks where different subtasks are given to various parts of the organization (divisions, departments, pools, lines and/or individuals) to work in parallel. Some tasks could even be completed by contract where the organization is not directly concerned about the nature or number of individuals that perform it. Moreover, an organization usually provides many standard *services* (like time keeping, information storage and retrieval, etc.) that are available on demand to improve its functionality.

In *Enterprise,* calls in one asset to other assets are translated into message communication. In dataflow models, the communication structure is directly specified by the dataflow graph. Since dataflow graphs are dense and complicated structures, they are unsuitable for specifying a distributed application. In *Enterprise*, these details are hidden from the user. Instead, the system, rather than the user, determines the flow of information and control in the application and inserts the appropriate communication code in the user's program. To identify the call structure, it is necessary to explore the implication of hierarchies in the *Enterprise* asset graph and to determine the call structure implied by an asset kind. The knowledge of the communication structure can be used to insert appropriate code into the user's module as well as processing procedure calls into message exchanges (Chan, 1992).

This section describes the communication structure of each asset and investigates the dataflow semantics of coercing an individual to each of the asset kinds. To avoid confusion, coercion is only allowed to apply to an individual, except to reverse the changes by coercing an asset back to an individual. Currently, *Enterprise* supports the following different asset kinds: individual, line, department, pool, contract, division and service.

3.3.1. Individual

An *individual* contains no other assets. An individual is analogous to an individual person in an organization. An individual executes its code sequentially to completion when called. The next call to the same individual waits until the previous call is completed. An individual may be called by any external asset using its name. An individual is equivalent to a sequential program.

As illustrated by the icon in Figure 3.1, the program executes as a single thread as illustrated by the single arrow. The single arrow also serves to indicate that the program expects an entry procedure at the next connecting node below it to accept a call made from the program. If the entire application is run sequentially, the graph will contain only one individual node, without any node connecting to its outgoing arrow.

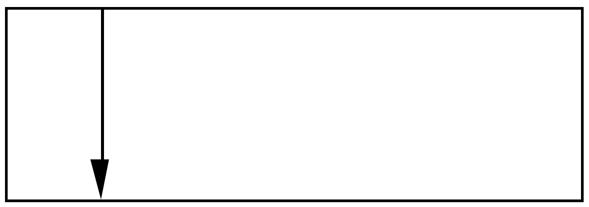


Figure 3.1. An Individual Asset

3.3.2. Line

A *line* consists of a fixed number of ordered heterogeneous member assets. Each member makes a call to the next member in the line. A line is analogous to a manufacturing, construction or assembly line in an organization. Each member of the line refines the work of the previous asset and passes the work on. For example, a line might consist of an individual who takes a pizza order, someone else to make the pizza and a third person to deliver it. Any subsequent call to the line waits only until the first asset has done its subtask of the previous call, not until the entire line has completed. The first asset in a line functions as the *receptionist* for the line and is the only asset that is externally visible. Therefore the first asset of a line is the only asset that may be called by an external asset and it shares its name with the line asset for this purpose. Lines are analogous to pipelines in the literature.

If an individual is coerced to a line, the original entry procedure resides in the first asset in the line. If the individual makes a call to an external asset, the call will be moved to the last asset in the line instead. As shown in its icon in Figure 3.2, the multiple arrows indicates that each node in the line expects an entry procedures at the next connecting node to accept a call made from it. Each node in the line will be inserted with code for establishing communication to the next node in the line. The number in the icon indicates the number of members in the line, which is 3 in this example.

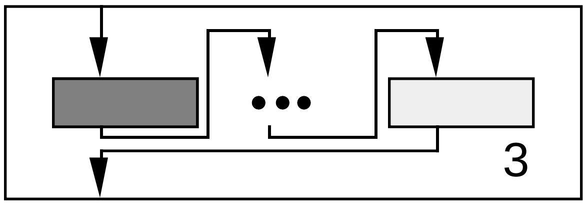


Figure 3.2. A Line Asset

3.3.3. Pool

A *pool* contains a fixed number of identical assets as shown in Figure 3.3. It is analogous to a pool in an organization where every member works on an identical task. For example, consider a pool of bank tellers. A client is services by the first available idle teller, and if all the tellers are busy, the client has to wait. Since pool members are externally indistinguishable (they perform the same task and share the same code), an external call cannot select a particular pool asset. Therefore all pool assets share the same name with the pool asset, and an external call addresses the entire pool. A pool is equivalent to a master-slave construct with a fixed number of slaves in the literature.

Since all members of a pool are identical to an external asset, in terms of control flow, a pool is similar to an individual. The flow of any outgoing or incoming call of an asset is not altered when an individual is coerced to a pool. However, since multiple replicas are ready to serve a call, some management is needed to direct the call efficiently to an idle replica. As in a line, the number in the icon indicates the number of members in the pool, which is 3 in this example.

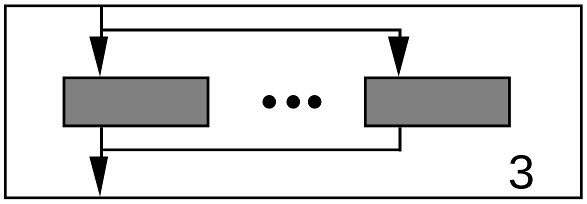


Figure 3.3. A Pool Asset

3.3.4. Contract

A *contract* contains a collection of identical assets, so it is similar to a pool (Figure 3.4). However, the number of assets in a contract is dynamic and depends on the number of processors that are free at any time. A contract is analogous to a contract that an organization lets to execute a collection of identical tasks. For example, an organization might contract out the package delivery job to a courier company. The courier company is called whenever a package delivery is needed. The organization is not concerned with how the courier company uses its resources, or the route chosen to deliver the packages. The number and capability of resources used by the courier company and the amount of competing traffic can affect the delivery time. Similarly when an *Enterprise* call is made to a contract, an idle asset executes the call. If all assets are busy and no more are available for hire, then the call waits for an asset to become available. A contract asset shares the same name with its members and the members all have identical code, as is the case with a pool asset. A contract is analogous to a dynamic master-slave construct, where the number of slaves varies in response to program needs (demand) and resource utilization (environment).

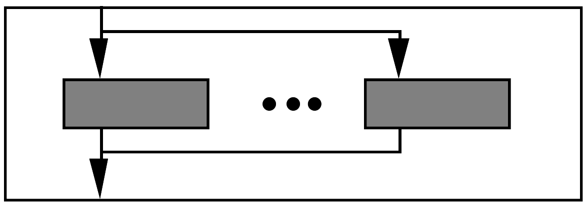


Figure 3.4. A Contract Asset

As mentioned above, a contract is the same as a pool except that the number of replicas varies dynamically. The icon of a contract is similar to that of a pool. The asset should be managed to direct the call efficiently to an idle replica and to vary the number of replicas dynamically.

3.3.5. Department

A *department* consists of a fixed number of heterogeneous assets. Every department has a single receptionist asset that shares its name with the department so that the department can be called by external assets. The receptionist makes direct calls to all the other assets in the department which then execute concurrently. This is analogous to a department in an organization where a receptionist is responsible for directing all incoming communication to the appropriate place. The department is similar to parallel constructs such as *fork/join* or *parbegin/parend* (Singh, Weber and Gupta, 1991; Dijkstra 1968), which creates parallel threads that execute independent code.

A department contains a fixed number of heterogeneous assets. The first asset in a department is the only asset that may be called from an external asset. It is also the only asset that make direct calls to all the other assets in the department. The flow of any outgoing and incoming calls to an asset are not altered when an individual is coerced to a department, except that the first asset will make calls to all the other assets in the department in addition to the outgoing call. The call structure is presented in Figure 3.5. The multiple arrows indicates that the first node in the is connected to several entry procedures in addition to the external node. The number indicates the number of assets contained. Communication to the external is only possible at the first node. The first node in the department will be inserted with code for establishing the communication paths described above.

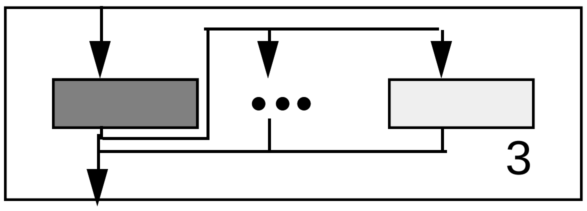


Figure 3.5. A Department Asset

3.3.6. Division

A *division* contains a hierarchical collection of identical assets with a fixed breadth and depth where work is divided and distributed at each level. Every division has a single receptionist asset that shares its name with the division so that the division can be called by external assets. Divisions can be used to parallelize divide-and-conquer computations.

In terms of call structure as shown in Figure 3.2, a division is a dynamic line in which every element of the line runs the same program on different data. Each element of the chain will recursively call an instance of itself. If the recursion reached a point in which distributed execution is not feasible (judging from the grain size or resource availability), a local call instead of a call to a remote process server is made, in which case the program is executed sequentially. Therefore, a division is also similar to a contract that calls itself recursively. A call made to a division must be coordinated to ensure that it will be handled by an idle server.

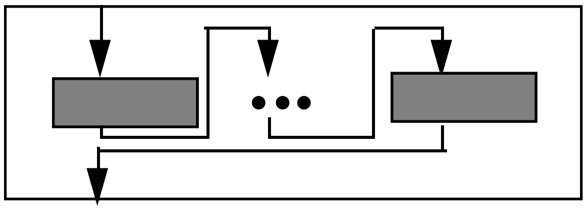


Figure 3.6. A Division Asset

3.3.7. Service

A *service* contains no other assets. However, unlike an individual that can be employed by a single asset, a service may be used by more than one asset at the same time. A service is analogous to any asset in an organization that is not consumed by use and whose order of use is not significant, for example, the clock on the wall. A service may be called by any external asset using its name. A services is analogous to a manager for shared memory (or 'tuple space') in the literature. Since a service may be called in any order by any asset, a service cannot be described by the *Enterprise* graph which is structured and hierarchical. Instead, services are considered as an unordered collections of modules external to the *Enterprise* graph.

A service is like an individual except it can accept calls from more than one external asset at one time. Since the call is from an external asset, from the view point of the service itself, it is identical to an individual. Every service can potentially be called by any other assets so the compiler inserts the code to establish connection with all the services in the code of every assets.

3.4. Parallelization by Coercion

Based on the above investigation on *Enterprise* assets, parallelism achieved by coercion in *Enterprise* can be classified as replication and partition. Replication splits the original asset into homogeneous assets and executes each asset on a different machine if possible. Since the assets have to be homogeneous, the call structure of the program is not altered after the coercion. Partition splits the asset into heterogeneous assets and changes the program's call structure. Coercion of an asset into a line or department results in new call structures. Each asset in a line contains a call to the next asset in the line. The first asset in a department calls all the other assets in the department. Every asset in a line or department after the partition will have the same replication characteristics as the asset before it was partitioned.

3.5. Conclusions

This chapter presented the *Enterprise* programming model. The new semantics of the procedure call introduced by *Enterprise* were illustrated. Procedure and function calls are processed into asynchronous remote procedure calls. For a function call, the program blocks when the data to be returned are accessed in part or in whole. This chapter also described in detail all the parallel techniques supported by *Enterprise* assets. Each asset represents a different way of replicating or partitioning an operation across multiple processes using analogies found in an organization. Moreover, the semantics of each asset was explained in terms of its data/control flow structure and its partition and replication characteristics.

Chapter 4

Programming in Enterprise

4.1. Introduction

The process of developing any application generally involves three steps: design the program, compile the program and execute the program. It may take many iterations through the above steps to debug and test the application before it is considered complete. In *Enterprise*, in addition to the program code, it is also required to specify an application graph. Constructing an *Enterprise* application involves the steps shown in Figure 4.1. The diagram also illustrates the dependencies and dataflow involved. An *Enterprise* application graph contains all the information as to how the program would be run in parallel. The program code looks the same as ordinary sequential code and is devoid of any parallel constructs.

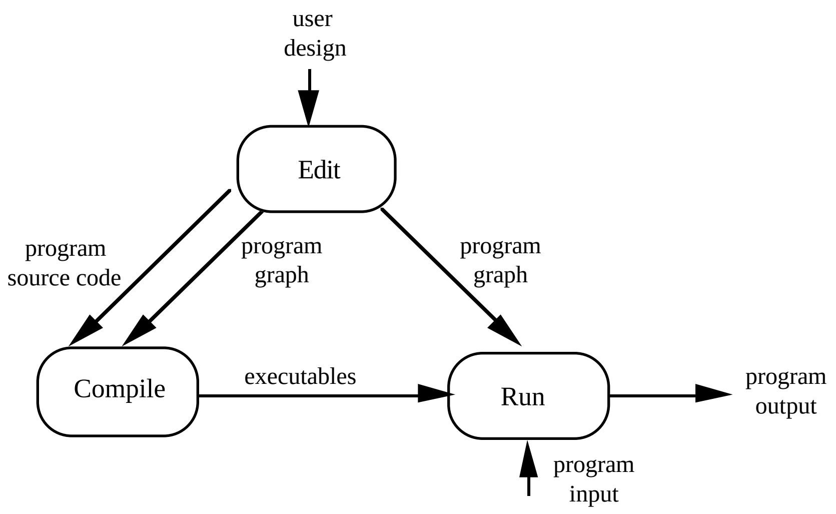


Figure 4.1. Major Development Flow of an *Enterprise* Application

This chapter presents the programming facilities in *Enterprise*. It contains a walk-through of an *Enterprise* session to construct a distributed program. This chapter does not intent to demonstrate the use of all the combinations of *Enterprise* assets, but serves to illustrate that the *Enterprise* model offers a simple and easy way to develop parallel applications. The focus is on *Enterprise* program design using the textual interface, which is part of the work of this thesis.

Program design using the graphical user interface can be found in other related publications (Chan et al., 1991; and Szafron et al., 1992).

4.2. Textual Interface

The textual interface was originally implemented as an interim measure to decouple the development of the other parts of *Enterprise* from the graphical user interface, which was not available. However, a textual interface also provides the user a handy way to develop *Enterprise* applications when a graphics terminal is not available or when X window is not accessible. A common scenario of the above is when the user is running *Enterprise* over a modem line.

In the graphical interface, an integrated environment is supported and the user compiles and executes an *Enterprise* application at the click of a button on the screen. In the textual interface, the system consists of three UNIX programs: the user's favorite text editor, such as *emacs* or *vi* , the program *compile*, and the program *run*. All three programs can be executed from the UNIX shell. As shown in Figure 4.1, the text editor is used to generate the program code and program graph, which are then used as input to the compiler to obtain the executables. *Run* uses the program graph to launch the compiled executables in a distributed environment.

In *Enterprise*, the program graph has two parts. The first part is a graph of assets arranged in a hierarchical order. The other part is an unordered set of service assets. Although other parallel programming environments support graphical views, as described in Chapter 2, these views are either non-editable or are edited by drawing nodes and arcs that represent processes and communication paths. In *Enterprise*, the user starts with an individual asset that encapsulates the entire program. As mentioned in the previous chapter, if an individual is used to run the code, it will execute sequentially on its own. The user introduces parallelism to the program by coercing the individual assets into assets that contain multiple processes, for example, contracts, pools, departments and divisions. The multiple processes contained in an asset are all individuals to begin with. The user splits the original program among the processes so that each piece of code can be run concurrently. More parallelism can be achieved by further coercion of the other individual assets into assets that represent multiple processes. For example, one can build a contract where each asset in the contract is itself a line of individuals. Services can be considered as a *global* computation facility (such as clocks) which can be used by any assets of other kinds and the order of use is not important. Therefore, services are organized as a set, separated from the main asset graph. An *Enterprise* diagram can be constructed using any combination of assets.

An *Enterprise* graph can also be viewed as a hierarchical structure of repetitive coercion applied to individual assets. The graph exhibits a structure similar to the top down decomposition of the program. The model allows the user to parallelize a program just by coercion. Coercing an asset from one kind to another *usually does not require any changes to the user's source code* (there may be few exceptions to this, particular if pointers are used to alias data). However, some gathering or separation of functions from one file to another may be needed. For example, if an individual is coerced into a line of three individuals, the code associated with the original individual would be separated into three files. Each individual in the line will execute a separate file.

In the graphical interface, it is possible to coerce an asset of multiple processes back to an individual and undo the parallelism introduced by a previous coercion. The user can thus experiment with parallelism of different grain sizes. However, in the textual interface, this backward operation is not a concern since the user can delete the unwanted coercion statement from the textual graph.

4.3. Syntax of the Textual Enterprise Graph

The textual *Enterprise* graph is organized as an ordered sequence of *coercion* operations and asset *attributes* specifications. As mentioned in the introduction of this chapter, the order of *coercion* operations is important since it represents the hierarchy of parallelism which leads to different parallel structures (such as *line in a pool* and *pool in a line*).

The operations and specifications are defined by *statements*. An *Enterprise* graph is specified by one or more statements. A statement is defined as a stream of characters ending with a carriage return. A statement may contain a number of *identifiers*. An identifier is a stream of characters ending with a space or carriage return.

An identifier that begins in the first column signifies that the statement specifies a *coercion* operation or asset *attribute* specification. If the identifier is one of the known assets (individual, line, department, pool, contract, division, and service), the statement is recognized as a coercion operation. Otherwise, the identifier is assumed an asset name and the statement is considered as an attribute specification. Figure 4.2 illustrates the syntax of coercion statements.

```
individual <name>
line <size> <name_0> <name_1> ... <name_size-1>
department <size> <name_0> <name_1> ... <name_size-1>
pool <size> <name>
contract <name>
division <depth> <breadth> <name>
service <name>
```

Figure 4.2. Syntax of Coercion Statement

The above figure serves only to illustrate the syntax of the *Enterprise* graph; the semantic implications of the graph will be discussed in Section 4.4. The first identifier represents the kind of asset (which can be one of individual, line, department, pool, contract, division or service) to be coerced. As discussed in the previous chapter, each asset has its own semantics which support a particular kind of parallelism. A user module is associated with an individual asset by default, so a statement that coerces an asset to an individual may be omitted. Following the first identifier is a list of arguments which gives the size of the asset, the name of the asset to be coerced, and the members of the asset.

The size argument is usually a number representing the degree of parallelism supported by the asset. Usually, the number (with the exception of division and contract) defines the number of processes used to run the code of the asset. When the asset has a default size, for instance, if it is a contract, service, or individual, the size argument is omitted. In contract assets, the number of processes used is varied dynamically at run time based on the number of available machines in the environment as well as the amount of work passed to the asset. In service and individual assets, the size is always one. Therefore these assets do not require the size argument. The division asset requires two size arguments: breadth and depth.

The identifier following the size argument gives the name of the asset to be coerced, which must be an individual. As mentioned in Chapter 2, in all *Enterprise* assets, the first member of the asset, also known as the receptionist, shares the name with the asset itself. In the textual graph, the name always refers to the individual asset when the name is shared. If the receptionist of the asset is an individual, the name refers to the receptionist. If the receptionist contains other assets, the name refers to the receptionist of the contained assets and so on.

If the assets contain heterogeneous assets (as in lines and departments), the identifiers following the name are the names of the other members of the coerced asset. As described in Chapter 2, the first member of an asset shares the name with the asset itself. That is also the name of the target on which the coercion is performed. The name of the first member is already given and therefore it is not required. Only the names of the other members are required as member arguments. In homogeneous assets such as individuals, pools and contracts, since all members are alike and share the name of the coerced asset itself, no member argument is required.

To summarize the arrangement of coercion operations in an *Enterprise* graph, an *Enterprise* graph is hierarchically organized. It always begins with the coercion of the asset running the main program and the application graph is constructed just by successive coercion. The main asset remains the root of the graph. Every asset to be coerced, with the exception of the main and service assets, must have already been specified as a member argument in some previously defined asset.

As mentioned in the introduction, services are arranged as a set of assets that can be accessed in any order by any other assets, and order is not important in specifying services. Every asset kind, except service, in an *Enterprise* graph obeys the hierarchical structure.

In addition to coercions, the *Enterprise* textual graph also contains information specific to the assets themselves, such as libraries to compile them with, and machines to run them. If an identifier that begins in the first column is not one of the known assets (individual, line, department, pool, contract, division, and service), the identifier is assumed an asset name and the statement is considered as an attribute specification. An asset must be defined in a coercion statement before its attributes can be specified in an asset specification statement, which is optional. Attributes are also optional. The user may leave out attribute specification entirely and the system then makes the decision when necessary. As with coercion, these attributes apply only to the individual asset. Attribute statements must begin with the name of the asset, and can be followed by one or more attributes which must be started with a *tab* character in the first column. These attributes specify user preferences such as machine allocation constrains and compilation libraries. Figure 4.3 illustrates the syntax of asset specification statements.

```
<name>
    library <file0> <file1> ...
    exclude <machine0> <machine1> ...
    include <machine0> <machine1> ...
```

Figure 4.3. Syntax of Asset Specification Statement

If the identifier after the tab is "library", the remaining identifiers in the statement give the libraries to be linked when compiling the asset.

The user can also specify a *machine preference list*, as one can do with *FrameWorks*. If the identifier after the tab is "include", the remaining identifies in the same statement gives the machines preferred to execute the program on. If none of the desired machines is available, any idle machine will be used. If the identifier is "exclude", the succeeding identifiers name the machines to be avoided. Even if the machine is idle, it will not be used. Since the main module is always run from the machine where the user's terminal is connected to, allocation constrains for the main module are ignored. The machine on which the user runs the main module cannot be excluded to avoid deadlock in allocation. This ensures that there will be at least one machine to put the processes on. The bottom line, when the entire network of machines are all busy, all the processes would be running on the single machine from which the user starts the *Enterprise* program. This is equivalent to running the program sequentially on that machine. The user may leave the allocation entirely to *Enterprise* by specifying no allocation constraints. *Enterprise* then selects machines from the environment based on their load averages.

The usage of CFLAGS is equivalent to that in *Make* (Feldman, 1979). It can be used to specify the path for linking '#include' files in a C program, as well as flags for optimization options. The specification may appear anywhere in the file, but must begin in the first column. The following figure illustrates the syntax of a CFLAGS statement:

```
CFLAGS = <flag0> <flag1> ...
```

Finally, comments may appear anywhere in the file provided that they begin with a '#' character in the first column. For example, the following are comments:

```
# This is a comment
# This is another comment
```

4.4. Writing an Enterprise Application

In this section, a walk-through of an *Enterprise* session to construct a distributed program is presented. A program that animates a school of fish swimming across a display screen is considered. The program involves generating a sequence of graphical images that, when shown in rapid succession, generate the illusion of motion. Producing these images involves computation intensive calculations and a high volume of data. There are three fundamental operations in the program: *Model*, *PolyConv* and *Split* with the following functionality:

- *Model*: Animates the movement of fish in a tank based on behavioral characteristics of the fish and the laws of physics. It determines the location and motion of each object in a frame, stores the results in a file, calls *PolyConv* to process the frame and proceeds to the next frame.

- *PolyConv*: Reads a frame from the disk file, performs some data format transformations, viewing transformations, projections, sorts and calls *Split*, passing it a transformed frame and a sequence number.

- *Split*: Performs hidden surface removal, anti-aliasing and stores the rendered image in a file.

This problem was contributed by a research group in our Department and is certainly more complex than portrayed by our brief description. The program is structured so that *Model* executes a loop that, for each frame in the animation, performs some work on the frame and calls

PolyConv with the results. *PolyConv* refines the image received from *Model* and calls *Split*. *Split* does the final polishing of the frame and writes the final image to disk.

An *Enterprise* program consists of a graph of assets and files of C code. As described in the previous chapter, each asset encapsulates a single C procedure/function, called an *entry procedure*, and a collection of support procedures used by the entry procedure, all in a single file. A program will consist of several assets. In this example, there will be three assets: *Model*, *PolyConv* and *Split*. Users who have access to a graphics terminal manipulate icons that represent assets. The graphical interface generates a textual file containing the graph of assets. Users who do not have a graphics terminal available to them can specify the textual file directly without using the graphical interface.

Users create the graph file with their favorite text editor. As a convention, the name of the program is used to name the graph file, *Animation* in this case. The design session begins as follows:

1) A new program consists of a single *individual* asset that represents a sequential program component. The code for the procedures *Model*, *PolyConv* and *Split* could be associated with this single individual asset and run as a sequential program. Every asset should have a name associated with it. In this case, since *Model* is the entry procedure (or the main routine), the asset should be named as *Model*. This program is represented by entering the following statement into the graph file:

   ```
   individual Model
   ```

2) Obviously, *Model* does not have to wait until *PolyConv* finishes the execution of the first animation frame to start working on the second frame. Similarly, there is no reason for *PolyConv* to wait for *Split*. Therefore, the asset can be *coerced* to (replaced by) a *line* asset by adding the following statement into the graph file.

   ```
   line 3 Model PolyConv Split
   ```

3) The statement above specifies that the asset *Model* is coerced to a line of 3 assets (which are *Model*, *PolyConv* and *Split*). By default, an asset that is not given an asset kind is assumed an individual. Statements such as

   ```
   individual Model
   ```

   are not required in the file. This statement can therefore be removed from the file.

4) Further, the *Split* asset can be coerced from an *individual* to a *contract* by entering the next statement:

```
contract Split
```

After this step, the graph file should look like the following:

```
line 3 Model PolyConv Split
contract Split
```

5) *PolyConv* can be specified to avoid the machine *sundog* but preferably use the machine *sass-lake* or *salt-creek*. Two attribute specifications should be inserted to make the file look like the following:

```
line 3 Model PolyConv Split
contract Split
PolyConv
   exclude sundog
   include sass-lake salt-creek
```

Asset attribute specifications can appear anywhere after the coercion statement that creates it. For example, the statement which creates the *PolyConv* and *Split* asset is the statement:

```
line 3 Model PolyConv Split
```

Therefore, the following variation of the graph file is equivalent to the original:

```
line 3 Model PolyConv Split
PolyConv
   exclude sundog
   include sass-lake salt-creek
contract Split
```

As shown above, an asset attribute specification should always begin with the name of the asset in the first column. It is then followed by one or more attributes. Each attribute begins with the tab character. The order in which these attributes are specified is not important. For instance, the 'include' attribute may come before the 'exclude' attribute without changing the semantics.

6) The *Split* asset can be specified to compile with the libraries *-lm -lx11* and *mylib* by adding the library attribute. After this point, the file should look like the following:

```
line 3 Model PolyConv Split
PolyConv
   exclude sundog
   include sass-lake salt-creek
contract Split
Split
   library -lm -lx11 mylib
```

7) Finally, comments may be added anywhere in the file, provided that they begin with a '#' in the first column:

```
# This is the Animation program graph.
# The application contains a line of 3 assets
line 3 Model PolyConv Split
# PolyConv should not run on sundog, but on sass-lake & salt-creek
PolyConv
   exclude sundog
   include sass-lake salt-creek
# Split is coerced into a contract
contract Split
# Split uses the library -lm -lx11 & mylib
Split
   library -lm -lx11 mylib
```

As illustrated above, in an *Enterprise* graph the length of the critical data path is represented by the number of coercion operations in the file, while the degree of replicated (non-line) parallelism is represented by the length (or number of assets) of a coercion statement.

The C code associated with an asset can also be entered using any text editor. The code should be named as the name of the asset with a ".c" suffix. In the example, the *Model* code would be saved in the file Model.c, illustrating the close relationship between *Enterprise* code and sequential C code.

If the C code for the individual *Split* contains a sequence of procedure calls (or even a single procedure call at the end of it), then *Split* could be coerced to a line by, for example, appending the following statement to the file (and prepare ".c" files for the two added assets):

```
line 3 Split HiddenSurfaceRemoval AntiAliasing
```

Each of the assets in the line would represent one of the procedure calls. Several other asset kinds are supported by *Enterprise* and they can be combined in arbitrary hierarchies.

4.5. Compiling an Enterprise Application

The program *compile* can be used to compile an *Enterprise* program. It takes a program name, which should be *Animation* in our example. For example, the following command is used to compile the *Animation* application:

```
compile Animation
```

Enterprise automatically inserts the code necessary to run the program in a distributed environment. The compiler then generates the executable and reports any errors to the user. This involves inserting code to establish remote procedure connections, and translating entry procedures calls to message communication. The latter is done by a modified GNU C compiler (Chan, 1992). In the above example, all the assets in the *Enterprise* program graph will be compiled regardless if they have been changed. The user can specify particular assets to be compiled and only those assets will be generated by the compiler. This is useful when the user has only changed some assets since the last compilation. The name of the assets to be generated are given as additional arguments on the command line. The order of the assets in the argument list is not important. For example, the following command is used to compile only the *Split* and *PolyConv* asset in the *Animation* application:

```
compile Animation Split PolyConv
```

An asset is required to be recompiled only when one of the following occurs:

1. its code has been changed, or
2. it has been coerced to a different asset, or
3. a service has been added or removed (in which case all the assets except services needs to be recompiled).

In the complete graphical interface, the application manager incrementally logs any change in the *Enterprise* program, both graph and source code, and determines which assets are to be recompiled. With the textual interface, the user would have to determine which assets are changed to take advantage of the partial compilation facility.

4.6. Running an Enterprise Application

After the program is compiled, the user can run the program. As with compile, run takes the name of the program as the required argument on the command line. For example, the following command is used to execute the *Animation* application:

```
run Animation
```

If additional arguments are given after the program name, they are treated as arguments to the program to be executed. For example, to run another *Enterprise* program *Primes* in which the sequential program takes the range of primes to be found, the following command is used instead:

```
run Primes 0 100
```

Redirection of UNIX standard input and output of the main module is supported in the same way as in a sequential program. *Enterprise* finds as many processors as are necessary to start the program, initiates processes on the processors, monitors the load on the machines and (if a contract is used) dynamically adds additional processors to the application based on the demand and supply of idle processors. The load averages and allocation of processes on the machines may be displayed at the same time when the program is executing. If insufficient processors are available to start the application, *run* will ask the user to select one of the following choices:

1. The program should be aborted.

2. The program should wait until sufficient processors are available and start the application then. The system monitors the load in the environment and automatically initiates the application when it is possible.

3. The program should start immediately with the currently available processors. If the user allows process migration, processes will be migrated to idle processors as they become ready. UNIX has no kernel support for process migration, and it is external to UNIX. In some cases, as when processes use files, process migrations may not be possible.

The program *run* also supports options to suppress process migration, to log flow of messages and processor allocation status to a file, or to start up a new window (if running under X window) to observe this information. These options are specified as command line arguments *before* the name of the program. For example, to run the application *Prime* with logging of messages and displaying of these information on a created window, the following command is used:

```
run -x -l Primes 0 100
```

The following is a list of execution options and their meaning:

-x A new monitor window is to be launched in X window to output allocation status and message logs. By default, no new window will be created and this information is directed to the UNIX standard output.

-t The application directs the allocation status and message logs to the device specified after the option. For example, the following command runs *Animation* and redirects the output to the terminal *ttyp1*:

```
run -t /dev/ttyp1 Animation
```

This allows users to use another window to display allocation status and message logs when they have access to more than one terminal. There are tools, such as *emacs* and *layers*, which can multiplex several virtual terminals on a single physical terminal. An alternative use of this option is to turn off the allocation status and message logs display by redirecting them to */dev/null*.

-m The application is to be run with process migration. UNIX does not have kernel support for process migration. To implement process migration, a new process is created on the destination machine and the original process is killed. The system does not guarantee that the application behaves deterministically and that variables are consistent when process migration is used. By default, the program is run *without* process migration.

-l The allocation status and message logs are saved in a file in addition to output to the terminal. The filename is the name of the application suffixed with ".log". For example, the log file of *Animation* would be *Animation.log*. The file will always be appended, instead of overwritten, if it is not empty when the program is executed. By default, no logging is performed if this option is not specified.

The user's current directory will be used as the working directory of the *Enterprise* program. All the *Enterprise* processes will be run from that directory, including the *Enterprise* system files such as managers for the processes and machines. *Enterprise* also looks for a machine database file 'mach_file' in that directory. In this file, a user specifies machines to run *Enterprise* processes.

4.7. Conclusions

This chapter describes the textual interface which users can use to edit, compile and run an *Enterprise* application. Users need only to specify a program graph of assets and attach source code to the assets in the graph. Starting with a graph which contains a single individual asset running the entire program, users introduce parallelism in the program by coercing repeatively the individual asset into another asset that contains multiple processes. Moreover, users can define modules to run as services which are organized as a bag of globally accessible computation facilities. Allocation constrains to run a module on particular machines can be specified. The syntax to express the above information in the textual graph format was illustrated via a walk-through to construct a real *Enterprise* application. This chapter also introduced the basic facilities to design, compile and run an *Enterprise* application. A discussion of all the facilities in *Enterprise* is presented in the following chapter .

Chapter 5

The Overall Enterprise Architecture

5.1. Introduction

This chapter presents the overall *Enterprise* Architecture. It contains a brief description of the current work involved in *Enterprise*. It also serves to address the position of the *executive* among the other components of *Enterprise.* An overview of the design of the system is provided, an overview of its implementation follows.

5.2. System Design

In *Enterprise*, distributed application programs are modelled after an organization, as the name suggests. To be consistent, the same analogy is found in the logical components of the *Enterprise* system itself. In the architecture of *Enterprise*, there are six logical components: an interface manager, an application manager, a code librarian, an execution manager, a monitoring/debugging manager and a resource secretary as shown in Figure 5.1.

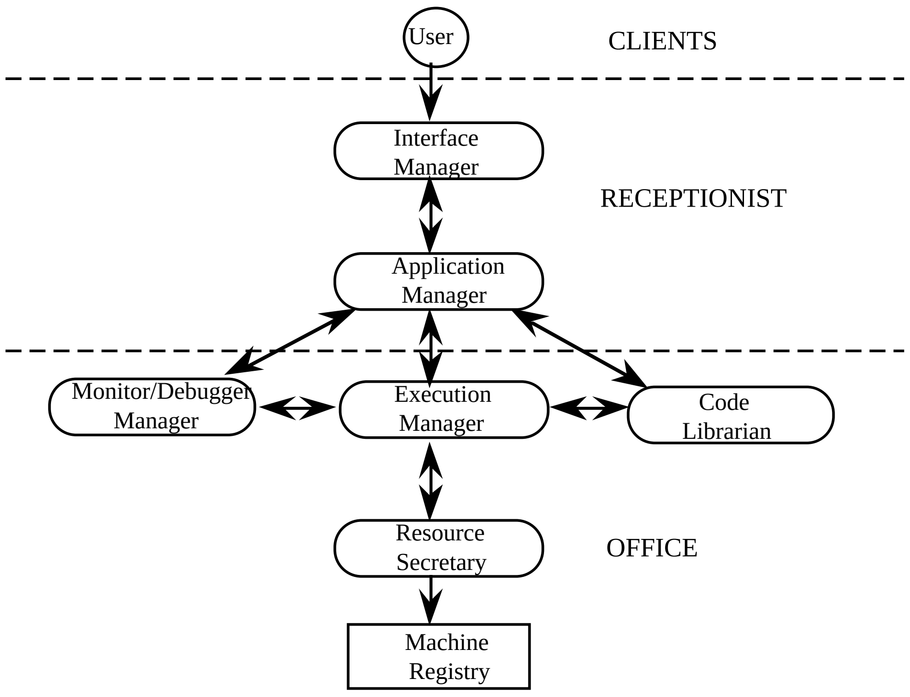


Figure 5.1. The Architecture of *Enterprise*

5.2.1. Interface Manager

The *Enterprise* graphical interface provides an environment for editing, debugging, compiling, configuring, executing and monitoring parallel programs. The user can develop parallel applications in a single unified programming environment. The tool features an asset graph editor for user to construct their applications using the organization analogy.

An object-oriented design was used for the user-interface. For example, each asset is an instance of an asset class and is responsible for knowing its name, attributes (like the length of a line asset or the size of a pool), components (like the components of a line), code, drawing itself, expanding itself, etc. Since different asset classes share many responsibilities, inheritance was used extensively. Moreover, the other interface components (windows, menus and dialogs) are also objects.

The current implementation of the user interface was written using the X Window System (Scheifler and Gettys 1986) on Sun workstations, making the interface highly portable. The implementation was developed in C++ using *Motif* (Young, 1990a; Young, 1990b). The combination was picked for easy integration with the rest of *Enterprise* (which is implemented in C), to remain faithful to the object-oriented design, to enhance portability and to minimize development time by utilizing the *Motif* class library of interface objects.

A graphical user interface provides a simple and powerful programming environment, however, a textual interface is also desired since a user may want to reconfigure a program when a graphics terminal is not available. The textual interface, of course, has limited capabilities compared to the graphical interface; for example the user may not be able to view the dynamic execution of a program.

5.2.2. Application Manager

The application manager is the control center for *Enterprise*. The application manager maintains all of the permanent information about an application, including the asset graph and the source code. The user can access application-specific information only through the application manager. Also, the manager ensures information provided by the user is correct and consistent with the current state of the program design.

5.2.3. Code Librarian

The code librarian manages the source and object code of different modules in a parallel application. Since *Enterprise* is aimed at a heterogeneous networked workstations environment, the librarian maintains multiple object codes for a variety of architectures. The librarian constructs and executes a *makefile* that is incrementally modified as the user changes the specifications of an application, and is parameterized to support different target architectures.

The librarian keeps track of the location of the source code and the corresponding object files for all assets in the application. Before an *Enterprise* application is started, the librarian ensures that all the required executables are ready and performs the necessary compilations for the unavailable executables on demand (Chan, 1992). Compilation requires knowledge of the kinds of assets involved, which is obtained from the application manager. During the compilation, appropriate *Enterprise* code is inserted into the module depending on its asset kind. Re-compilation of an asset is only required when either the user modifies the asset's code, the asset has been coerced to a different kind, or when the asset is to be run on a machine for which an executable is not available.

Requests for compilation come from two sources. First, the user can ask the interface manager to compile the application to reveal syntax and semantic errors. Second, the execution manager can request executables on demand at run-time. As machines become available, the code librarian informs the execution manager if an executable exists for that machine. If the executable is not available, the execution manager makes the decision as to whether to request the librarian to provide it or not.

5.2.4. Execution Manager

The *Enterprise* execution manager controls the execution of an *Enterprise* application. It creates all the processes specified by the asset graph, and sets up the required communication channels. It manages machine resources and migrates processes from busy machines to idle machines. It mediates communication among processes so that work will be directed to an idle process. It traps interrupts and faults in the execution and cleans up any run-away processes after the execution is done or aborted.

The decision on where a process is to be executed is made by the execution manager, implying that executables are general and do not have a specific machine name compiled into them. Associated with an asset is a *machine preferences list* which specifies any constraints on the machines to be used to run this asset. The default is that the program will run on any machine. The user, however, may choose to constrain the choice of machine in some way, say by execution speed or physical location. The method used to select machine preferences is similar to the technique employed in *FrameWorks*.

5.2.5. Resource Secretary

The resource secretary maintains the *machine registry*, which is a database of specifications on each machine in the network, such as the machine name, its architecture and type, operating system version, compiler name and options, a speed rating, RAM size, etc. The secretary determines periodically (a system adjustable parameter) the load on each machine and reports any change in a machine's status (from idle to busy or vice-versa) to the execution manager.

Each machine in the network has an associated ".enterprise" file in which the owner of the machine can specify when *Enterprise* programs are allowed to run. The default is to allow programs to run on evenings and weekends only.

5.2.6. Monitor/Debugging Manager

In parallel programming environments, monitoring and debugging are important facilities. They should enable a user to determine the bottleneck in a program and to recognize possible concurrency problems by monitoring the program's execution. *Enterprise* supports monitoring by program animation. Every call to an asset is time stamped and the information is either sent to the interface (to display dynamically the program's execution in real-time), or saved in a file for the user to replay the program at leisure. The user can use the debugger to step through an application at the asset call level and to set breakpoints any time an asset is called.

5.3. System Implementation

FrameWorks was built on NMP (Marsland, Breitkreutz and Sutphen, 1991), but its many limitations forced us to consider alternatives. *Enterprise* uses the ISIS package to do all the low-level communication. ISIS was chosen to remain faithful with the object based design philosophy of the system, and for easy integration with the rest of *Enterprise*. For instance, the user interface was designed using an object-oriented approach and implemented using C++. ISIS provides the necessary library routines to structure an object based application (Birman et al., 1985) on top of ordinary C code. Major portions of the system have been implemented using existing C programs, for instance, the compiler and the user's code have been written in C. ISIS provides global naming, message passing and process grouping mechanism for communicating processes (Birman, 1991; Birman, Cooper and Gleeson, 1991). Moreover, it also supports fault tolerance.

Enterprise is implemented as an ISIS program. Not only are the executables that *Enterprise* produces ISIS programs, but the architecture of *Enterprise* is designed as a multiple-process program that communicates using ISIS. Most of the components of the architecture described in Section 5.1 are communicating processes. The Application Manager is conceptually a separate process, but for efficiency is implemented as part of the Interface Manager. For a similar reason, the resource secretary is implemented as part of the Execution Manager.

5.3.1. Fault Tolerance

ISIS has extensive support for fault tolerance (Birman, 1992). Consider the case of implementing an individual. One approach for fault tolerance supported by ISIS is redundant computation. Several processes undertake the same task, with the caller waiting for the first response and continuing to execute without waiting for a response from the others. It achieves fault tolerance at the cost of wasted CPU cycles.

Another approach in ISIS is the coordinator-cohort computation (the standby approach) (Budiraja et al., 1992). The method works by ranking the members of a process group and then labeling the lowest ranking member as the coordinator for a request. This process will execute the request and broadcast a reply to the caller. The other members are cohorts; they are passive unless a failure prevents the coordinator from terminating normally, in which case they take over one by one, in rank order. The coordinator is also able to send the cohorts a copy of the answer returned to the caller at the termination of the computation. ISIS does not add extra cohorts to a coordinator-cohort algorithm when it is already running, so fault tolerance is limited by the number of cohorts at the start of the computation. If several requests arrive concurrently, the job of being coordinator will be split over the members of the group in a uniform manner. Thus a single process may be the coordinator for one or two requests while being cohorts for others. Moreover, there may be several coordinators at one time for different requests. The scheme thus exploits the distributed processing power of the group in a fault tolerant way.

If the asset is a contractor or a department, when some of the processes fail to complete, the number of replies will be less than the number of broadcast messages sent. In this case, the program needs to either recompute the missing piece, or reissue the entire request. Using this approach, the application handles the recovery from fault.

Both the replication and standby (coordinator-cohort) approaches achieve only partial failure resiliency. Computation results will be lost when the failure involves all the processors responsible for the computation. In ISIS, a tool is provided to log a process' events and, when a total failure occurs, to recover the state it was in from the log. This is done by logging a copy of the checkpoint of the computation state onto stable storage.

5.3.2. Process Migration

Migrating a process from one machine to another involves (Eskicioglu, 1990):

1) suspending of the process on the source machine,

2) transferring of the process state to the destination machine, and

3) resuming its execution on the destination machine.

These operations require support from the operating system. The initial implementation of *Enterprise* has been done under the Sun OS operating system, which does not offer any process migration facility. However, the development of *Enterprise* under an operating system which offers process migration, such as the V-System (Cheriton and Zwaenepoel, 1983), would allow for better load distributions. Under Sun OS, when it is necessary to relinquish a processor, it is sometimes possible to suspend the process in the middle of an execution and restart it over again on another machine (although, obviously, some work will have been wasted). Also, in the case when the workstation is being used as an employee of a contractor, it can be released when the employee completes the unit of work assigned to it by the contractor.

5.4. Conclusions

This chapter provides an outline of the *Enterprise* architecture, the tools to implement it, and some issues to be resolved. This includes a brief introduction of the major components of the system and their implementation, and the major issues like distributed debugging, fault tolerance, and process migration. This chapter was an introduction to the internal details of the *Enterprise* environment. It served as a bridge between the user-interface aspect of *Enterprise* described in the previous chapter and the operating system aspects described in chapters that follows.

Chapter 6

Design of the Enterprise Executive

6.1. Introduction

The *Enterprise executive* controls the execution of an *Enterprise* application. It is responsible for creating all the processes specified by the asset graph, and setting up the required communication channels. It manages machine resources and migrates processes from busy machines to idle machines. It mediates communication among processes so that work will be directed to an idle server process. It traps interrupts and faults in the execution and cleans up any run-away processes after the interrupt or fault.

This chapter describes the design and architecture of the *Enterprise* executive. The object-oriented design paradigm is introduced and some terminology used in this chapter are defined. The chapter then illustrates how the architecture of the executive was influenced by the object-oriented design methodology. This includes a discussion of the *Enterprise* module call protocol, implementation of different kinds of *Enterprise* assets, management of machine resources, and design of the execution shell.

6.2. Object-Oriented Design Techniques

Using an object-oriented approach, the system is structured around the *objects* that exists in our model. An object is an entity that has private data (*state*) and a set of operations (*behavior*) to manipulate that data (Stroustrup, 1988). To invoke the operation in an object, a *message* is sent to the object. The message contains a *selector* to invoke the specific operation (known as a *method*) in the object. Additional arguments may be sent with the selector in the message. A method is similar to a procedure and can only be invoked when the object receives the message whose selector corresponds to that of the method.

Another feature of object-oriented programming is the concept of a class. A class describes the common behavior of a collection of objects. An instance of a class is an object that has its own state. Class descriptions usually contain variables and the method that manipulates the objects. It is a mechanism to encapsulate the state and behavior of a set of objects. Classes are related to each other through an inheritance hierarchy. Through inheritance, objects in different classes share behavior. A subclass inherits state and behavior from its superclasses, and can override the superclass' behavior method by redefining the method.

Object-oriented design involves the following major steps, (Booch, 1986):

1) The objects in the system are identified. This involves the recognition of the role of an object in the model, for instance, whether it is a server or client. Typically, the objects identified are derived from the nouns which are used in the problem description. Also, a class hierarchy for similar objects is derived.

2) The operations required of each object are identified. The behavior of an object when invoked by messages from other objects is defined.

3) The interface and dependencies of each object are identified. This involves identifying the external views and interaction protocols among objects and classes of objects. This step defines the selector and message format among the communicating objects.

4) The objects are implemented. An object can be implemented by building on top of existing lower level objects or classes, as well as by decomposing it into subsystems of objects.

Object-oriented design is considered by many as a powerful methodology for developing applications. The approach is used in designing many parts of the *Enterprise* system, including the program model, user interface and the execution manager. Object-oriented programming is designed to produce modular, reusable, modifiable code. During experimentation with different implementation approaches, the ease of changes and enhancements to the existing code proved to be highly desirable. The ability to create subclasses of existing classes was extremely useful in reducing both the design and the development time of *Enterprise*. The following section outlines the overall architectural features of the executive and overviews the function of each component.

6.3. Overview of the Executive Architecture

Using an object-oriented approach to design the system, the objects that exist in the environment and their relations are identified. Responsibilities common in different assets are factored out and classified to achieve code reuse. The fundamental building blocks of the *Enterprise* executive can be summarized by the inheritance tree or class hierarchy in Figure 6.1.

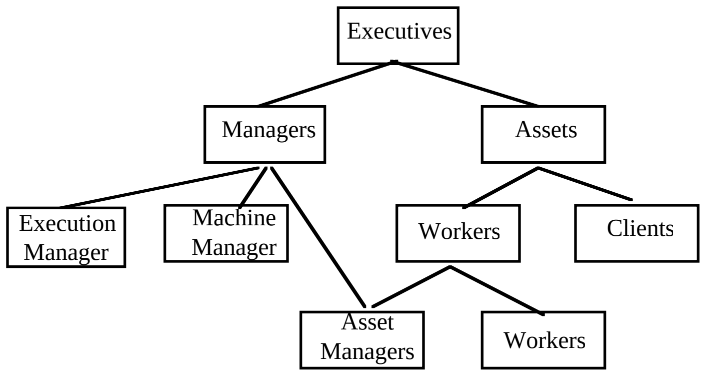


Figure 6.1. Class Hierarchy of *Enterprise* Executives

At the root of the class hierarchy is the **Executives**. This class encapsulates the common behavior of every asset in the system. Similarly, the **Managers** class defines the common behavior among objects which manage resources. The **Assets** class encompasses the common behavior of both **Workers** and **Clients** classes. The managers class encapsulates the shared characteristics of three subclasses, **Execution Manager**, **Machine Manager** and **Asset Managers**, based on the type of object being managed. The use of plural in the class name signifies that there are multiple run-time object instances for the class.

The managers manage the run-time resources of an *Enterprise* program. The execution manager interfaces the *executive* to the other components of the overall *Enterprise* architecture, such as the code librarian and application manager. The machine manager places *Enterprise* objects, both managers and assets, on idle machines based on their demand and availability. Asset managers select idle workers to execute the call initiated by clients, and requests resources from the machine manager to execute the workers. Every asset has an associated asset manager. An individual asset is also the manager for itself.

Assets are processes of modules of an *Enterprise* program. It is similar to a user program in a sequential programming environment, except that the user code is encapsulated by a layer of kernel structure to handle communication and synchronization. This layer is attached to the user code automatically by inserting code which handles communication and synchronization using a C language compiler. Figure 6.2 illustrates this concept of adding a layer of kernel to the user code.

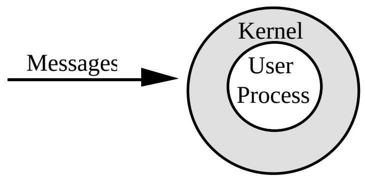


Figure 6.2. A Kernel Layer Encapsulates User Code

Assets may be viewed as a client or a worker. Worker assets are those which execute calls directed to them and client assets are those which initiate the calls. A worker asset can be further classified as a contract, pool or individual, according to its number of instances at run-time.

Figure 6.3 gives an overview of the interactions among different components in the *Enterprise* system when a program is executed. This collaboration of different objects in the system will be discussed in detail in the following sections.

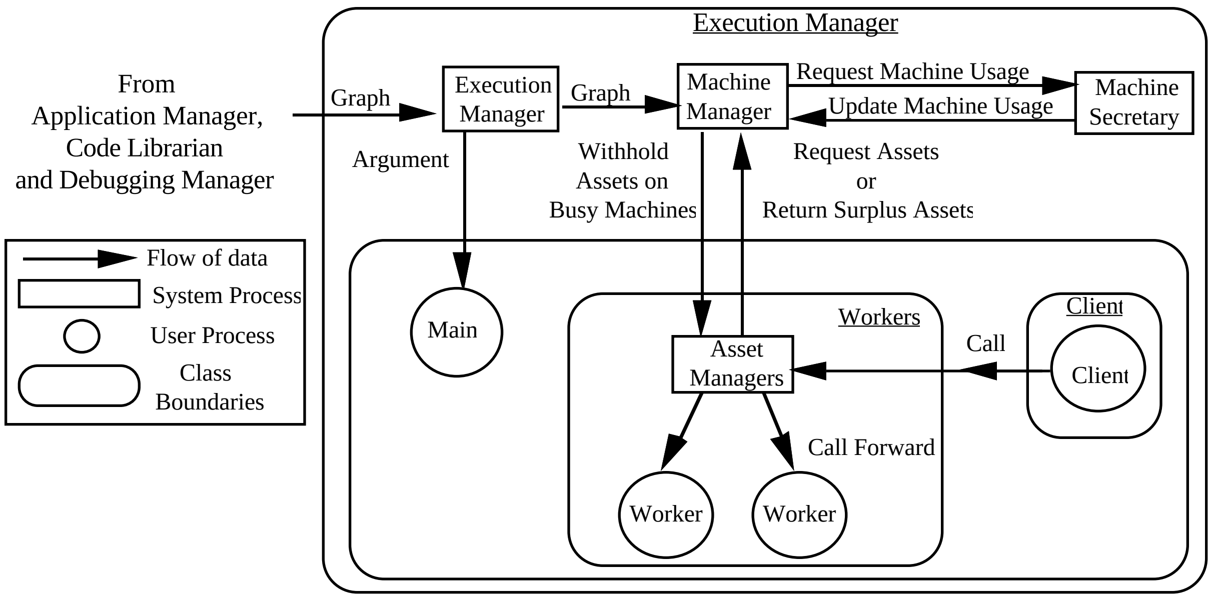


Figure 6.3. Overview of the Executive Objects and their Interactions.

6.4. Assets

The object-oriented design approach begins with identifying objects in the designer's model of reality. From the user's perspective, the only objects are the processes of the *Enterprise* assets in execution. Every asset in an *Enterprise* application can potentially function in the dual role of a client as well as a worker. Assets repeatedly serve calls placed to them by assets connected to their input and, therefore, function as workers to other assets. During the period of working on these calls, an asset may call other assets and thus may become a client. This behavior is common for all *Enterprise* assets regardless of their asset kind.

An *Enterprise* program at run-time can be viewed as client and worker(s) communicating using a call structure similar to that of a Remote Procedure Call (RPC) (Birrell and Nelson, 1984). There may be multiple instances of workers servicing a call from the client. The instance of workers may be fixed (pool asset) or varied (contract asset). A worker with only a single instance is equivalent to an individual asset.

6.4.1. Clients

Before a client call its workers, it must be acquainted with them, as discussed in the previous chapter. The client's potential workers can be determined from the call graph and code to establish connection with the workers can be inserted to the user's program. The call to an entry procedure is converted into a RPC. Depending on which procedure call is invoked, workers providing the corresponding service will be sent the RPC.

6.4.2. Workers

The worker has to announce its name on the system so that other modules can use the remote procedure it supports. It needs to determine its instance number if it has multiple copies of itself and registers itself (by its instance number) with its manager when it becomes idle. Using the convention of ISIS and many other distributed systems, a process can be uniquely referenced by its name together with its instance number. The number of instantiations that a worker has can be determined from the call graph and the required code to announce one's identity can be inserted into the user code. The entry procedure is translated into a message handling entry which extracts the call parameters from the messages. Every worker runs in an infinite loop to consume incoming calls, therefore, every worker process needs to declare a procedure which, when invoked, terminates itself. This procedure will be called when there are no more calls to be handled. It may also be called as part of the function to perform process migration. The system "migrates" a process by terminating it and restarts it on a different machine. Before termination, a worker needs to ensure that there is no unconsumed message outstanding in the buffer. However, sometimes it is desirable to terminate the application immediately before completion, as when a fault occurs in one of the modules, and the system should support this kind of worker termination as well.

6.5. Design Models and Their Communication Protocols

Using an object-oriented approach, the system is structured around the *objects* that exists in our model. By taking a different view point and by focusing on different aspects of the system, various interaction protocols between client and worker assets can be realized. Several approaches to design the application have been investigated. They involve trade-offs between flexibility and reusability of the design, efficiency of the implementation, and capability of correctly implementing all the *Enterprise* assets.

In this section, several models are presented which are different for the above consideration. The first model to be considered is the *restricted model.* This is the same as the restricted model used in implementing FrameWorks. Assets in this model communicate directly with each other. This model is simple but has severe limitations in ensuring fairness and mutual exclusion of handling calls among assets of multiple processes. Therefore, this model is only used to implement all the assets of a single instance in *Enterprise,* for example, individuals and services. The second model, the *reference model,* requires a client to acquire the right for exclusive use of a worker from an external process before communicating to the worker process directly. In other words, the external process *references* an idle worker to the client. The third model, the *forwarder model,* the client simply makes the call to the *forwarder* which then acts on behalf of the client to ensure that the call will be made to an idle worker. All calls go through the forwarder and so requests can be serialized. There are merits and drawbacks with each model and these will be discussed in detail later in this chapter.

6.5.1. Restricted Model

In the restricted model, the assets communicate with each other directly according to the call structure. This model is similar to the restricted model described in (Singh, 1992). Although the restriction of the model experienced in *FrameWorks* is different from that in *Enterprise,* the same name is chosen for historical reasons and the new model does have restrictive uses. From the user's point of view, the procedure calls and returns are the only messages exchanged among the assets. Hidden from the user, there is also the exchange of control messages to ensure proper synchronization and scheduling of the calls. An asset communicates this information with its called asset directly. For example, Figure 6.4 illustrates the communication between two processes in which A calls B.

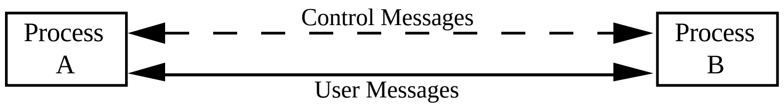


Figure 6.4. Direct Communication

The calling module is responsible for selecting an appropriate process to serve the call when more than one process is available (as in the case of a contract or pool). The advantage of this model is that it is simple, requires few processes to implement, and involves few message exchanges. However, this implementation model fails to handle the case when multiple clients are contending for multiple workers. The interaction relation is shown in Figure 6.5.

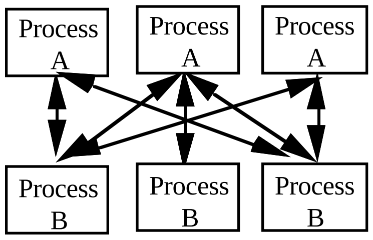


Figure 6.5. Many to Many Call Relation

The selection should be mutually exclusive to avoid multiple clients from calling the same idle worker simultaneously. The restricted model is limited in that clients do not share information among themselves as to which server is selected. To ensure mutual exclusion, two approaches can be used: locking the servers, or serializing the requests for servers. In both cases, an (conceptually) external process, a manager, is required to maintain the locking or to serialize the requests. Effectively, the original many-to-many relation is transformed to a pair of one-to-many relations with the introduction of this manager process. The transformed relation is illustrated in Figure 6.6.

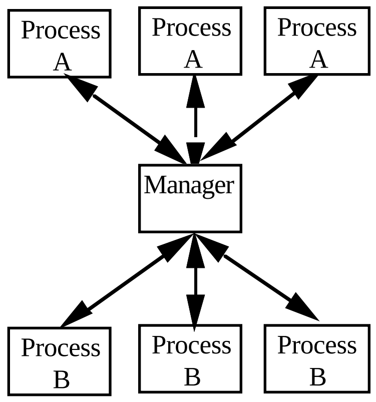


Figure 6.6. Transformed Relation by introducing a Manager

This process compensates the limit in the original model by its global view on the interaction which can ensure that only one client is accessing the worker at a time. In the original model, clients are limited in their perspectives. They do not know when others are calling a worker and which worker is being called. At this point, it can be concluded that the restricted model is insufficient to implement the *Enterprise* system when processes of multiple instances are involved. However, because of its simplicity, the model is still useful for implementing assets of a single instance, for example, individuals and services.

By using different approaches to serialize the requests for an idle worker, at least two new models for implementing pools or contracts can be realized. In the first model, the *reference model*, the management process just serves as a referral agency and a client still communicates directly with its worker once it knows which worker is available. In the second model, the *forwarder model*, the management process represents the worker. All communication is routed via the management process and the worker is hidden completely from the client. The reference model is described in *FrameWorks* as the *general model.*

6.5.2. Reference Model

Using this approach, before making a call, a client obtains the name of an available worker from the worker's reference. The reference marks the worker as unavailable once the worker has been referred to a client. The client then calls the worker directly and sends it the message containing the call parameters. When the worker finishes with the call, it replies directly to the client and registers itself as *available for employment* with its reference. Since all clients must check out the workers from the reference before making any call, requests for workers can be serialized and it can be ensured that workers are given to only one client at a time. When there is no available worker, the client's request for reference would block. Since in *Enterprise* all calls should be non-blocking, the entire task of asking for reference and making the call can be forked off to run asynchronously. In this way, the client would not block but continues with its execution. Figure 6.7 shows the flow of messages among the worker, client and reference.

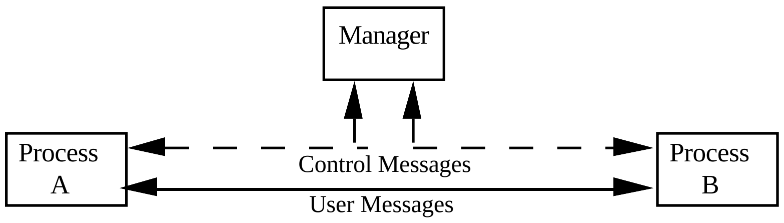
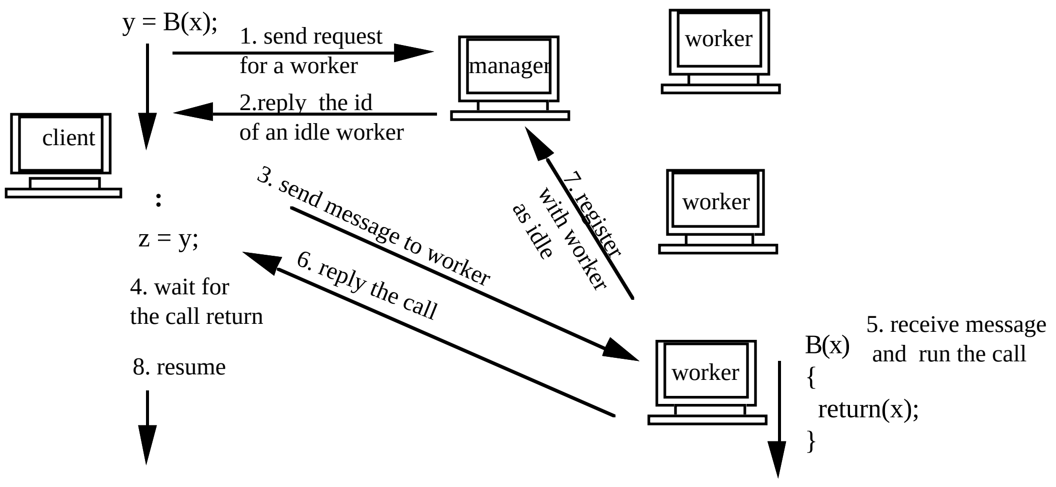


Figure 6.7. Manager Assisted Communication


y = B(x);
1. send request for a worker
manager
worker
2.reply the id of an idle worker
client
:
z = y;
3. send message to worker
7. register with worker as idle
worker
4. wait for the call return
6. reply the call
5. receive message and run the call
B(x)
{
return(x);
}
worker
8. resume

1) Client (A) sends a request for idle worker to the manager of the workers (B's manager), and waits for a reply.

2) The manager replies the name of the idle worker. If none is available, the request is put on a queue and will be met when the resource conditions allow.

3) Client (A) then puts the call parameters in a message and sends it to the idle worker. The message is sent asychronously and the client proceeds with other tasks.

4) If the return variable is accessed, the client (A) *waits* for the variable to be returned from the call.

5) In the meantime, the idle worker running B receives the message, interprets the arguments, and executes the procedure.

6) When the worker has done with the procedure, it puts the result in a reply message, and sends the message to A.

7) The worker sends a message to its *manager* to signify that it becomes idle. The manager of B receives the message containing the id of the idle worker. If the manager is not waiting for an idle *worker,* the id is put into the queue of idle workers.

8) Client (A) gets the result from the *reply* message (from B) and resumes.

Figure 6.8. The *Enterprise* Module Call Sequence for Manager Assisted Communication

Using the convention used in *FrameWorks*, the interaction structure is called Manager Assisted Communication since the manager plays a supporting role in the call protocol. The communication protocol for the reference model for a client and a pool of workers is illustrated in Figure 6.8. Messages are marked by arrows, showing the direction of the exchange. Control messages are represented by dotted arrows.

6.5.3. Forwarder Model

In the reference model, a client communicates with the worker directly. However, prior to making the call, the client must exchange messages with the manager of the worker. The forwarder model relieves the client from the task of finding (and even waiting for) an idle worker. The client can simply make the call and the forwarder is responsible for receiving the message from the client, selecting an idle worker, and sending the message to the selected worker. Neither the client nor the worker is aware of the existence of the other party. The forwarder interfaces all communication between them. As in the reference model, the forwarder marks the worker as unavailable once the call has been forwarded to a worker. When the worker finishes with the call, it also registers itself as *available for employment* with the forwarder. Since all clients must route their calls through the forwarder, requests for workers can be serialized and it can be ensured that workers are given to only one client at a time. When there is no available worker, the forwarder blocks but the client continues. The forwarder waits for an idle worker on behalf of the client. From the client's perspective, the call is non-blocking, and therefore satisfies the requirement of non-blocking call in *Enterprise*. Figure 6.9 shows the interaction among the worker, client and forwarder. The interaction is called Manager Delegated Communication since the client leaves the job of calling a worker totally to the manger. Figure 6.10 shows the sequence of events that occurs when an *Enterprise* call is issued from a procedure A, to call procedure B.

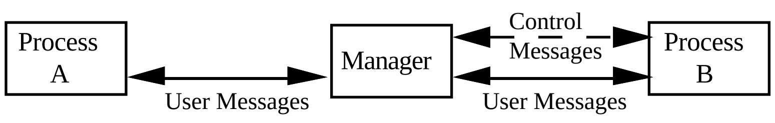
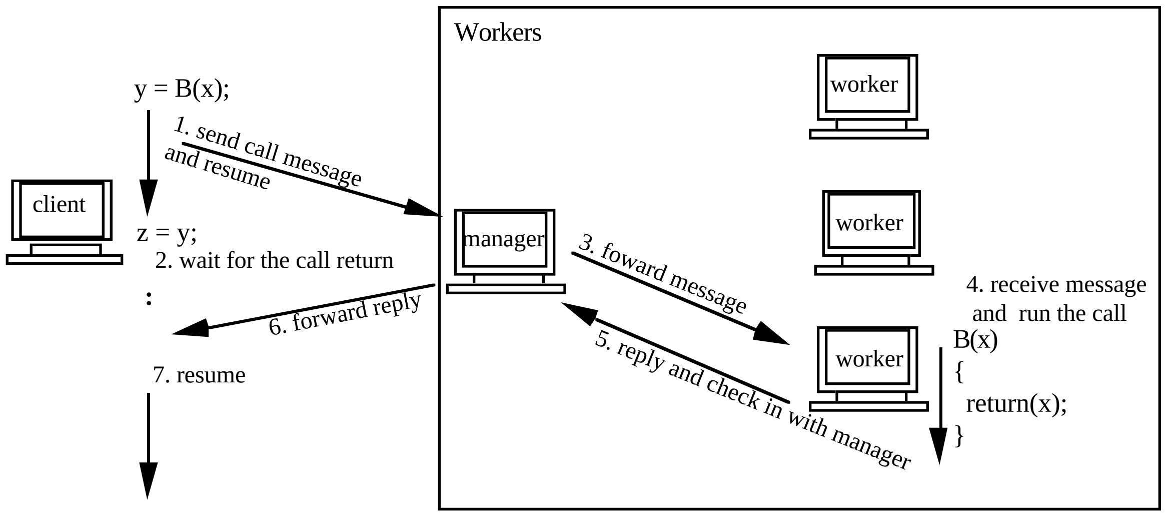


Figure 6.9. Manager Delegated Communication


Workers
y = B(x);
1. send call message and resume
client
z = y;
2. wait for the call return
:
6. forward reply
7. resume
manager
3. foward message
5. reply and check in with manager
worker
worker
worker
4. receive message and run the call
B(x)
{
return(x);
}

1) Client (A) sends a message containing the procedure name and arguments to the *workers* handling that procedure (B workers). The client (A) then resumes execution immediately.

2) If the return variable is accessed, the client (A) *waits* for the variable to be returned from the call when accessed.

3) In the meantime, the manager of B receives the message, selects an idle worker and forwards the message to the chosen *worker*. If no idle worker is available, the manager waits until one is available.

4) The selected worker receives the message, interprets the arguments, and executes the procedure.

5) When the worker has done with the procedure, it puts the result in a reply message, and sends a message to its manager to signify that it becomes idle.

6) The manager of B receives the message containing the id of the idle worker. If the manager is not waiting for an idle worker, the id is put into the queue of idle workers. The manager also forwards the reply back to caller (A).

7) The client (A) gets the result from the *reply* message (from B) and resumes.

Figure 6.10. The *Enterprise* Module Call Sequence for Monitor Delegated Communication

6.5.4. Comparative Analysis of Different Models

The two approaches described above to resolve the limitation in the restricted model are functionally equivalent. They differ only in their cost of development and cost of execution. Usually, when designing other systems, the issue of cost of execution are to be considered later at the implementation time. However, when designing an environment for distributed computing where high performance is desired, the designer should be aware of execution issues in every step of the design and implementation. To keep the design general, the consideration is made independent of any specific underlying communication system like ISIS or NMP.

The main advantage of the forwarder model over the reference model is the high level of transparency it supports. In terms of software maintenance, in the forwarder approach, assets are self contained. Changing the asset to a different kind affects only the asset itself. The logical client / worker relations possible in *Enterprise* are one-to-many, many-to-one, and many-to-many. The one-to-many client / worker relation is found in a client calling a worker which has multiple instances (as in the case of calling pools and contracts). The inverse is true when a client with multiple instances calls a single worker (as in individuals). However, with the forwarder in place, from both the client's and worker's perspective, there is only one interaction relation. The communication structure is one-to-one with the forwarder in both cases. As shown in the protocol, to the client, the manager and workers can be encapsulated in a single unit and treated as a black box. However, the interaction style cannot be well abstracted in the reference model. Since the caller is not concerned about the component of the worker in the forwarder model, coercing an asset to different kind involves only changing the asset alone and nothing else. This allows quick and easy experimentation of various parallelization techniques.

In terms of execution, in the forwarder approach, assets are not self-contained. The state of an asset is not only maintained by the caller and the called asset, but is also maintained in part by the forwarder. In the reference model, the messages are either in the client or in the worker. However, with the forwarder model, at any time the forwarder may have messages that are waiting to be delivered to the worker. Therefore, it would be more difficult and costly to make the system fault tolerant. There will be more complicated possibilities to consider when the system crashes to keep the assets in a consistent state.

As illustrated in the figures showing the message exchange protocols of the reference model and the forwarder model, both approaches require the same number of message exchanges for each procedure call made. However, the message sizes involved in the forwarder approach would be larger in general.

In the reference model, the worker sends a message to the manager to register itself as idle before a call can be made to itself. To make a call, the client sends a message to the manager to request a worker, and then makes the actual call and sends a message containing the call parameters to the idle worker. If the call is a function call, there will be an additional message containing the return parameters from the worker. Therefore, the message cost associated with a procedure call would be four exchanges in total. The cost of a function call would be five messages. Three of the messages are control messages for selecting an idle worker, and they are generally just a message containing a single integer.

In the forwarder model, the client sends the message containing the call parameters to the manager, who then forwards it to the idle worker. For each call, at least two messages will be required to get the call to its destination. If the call is a function call, two more messages would be needed. As in the reference model, the worker also sends a message to the manager to register itself as being idle at the end of each call served. Therefore, making a procedure call takes three messages in total. The cost of a function call is five messages. Only one of the messages is control messages for selecting an idle worker, which is also a message containing a single integer. The other messages contain call parameters, which, in general, may be of considerable size.

Sending an integer on the network costs the same as sending a packet (currently, *Ethernet* runs in chunks of 1 Kbytes) and so a small message size does not always imply substantial savings. It is equally important to minimize the number of message exchanges as well as the size of messages. When the total size of call parameters occupies less than a packet, since in the forwarder model fewer message exchanges are involved, the forwarder model may be more efficient. When the call parameters are of considerable size, it may be expensive to forward every call and calling the worker directly using the reference model would be more economical.

Initially, the design was based on the reference model. It was used in *FrameWorks*, so it was well understood and enabled a prototype to be rapidly constructed. However, as discussed in this section, the model is not very easy to maintain, since the implementation of an asset is dependent on the asset kind of all its called assets as well as its own. This is undesirable because *Enterprise* is an on going project in which new asset kinds have been continually being researched and old assets are being redefined. In the forwarder model, any change in the asset's implementation affects the asset alone. Therefore, the current approach uses the forwarder model.

6.6. Managers

Following the above investigation, it was realized that an object external to the call structure is required to maintain the call to ensure correct synchronization of procedure call messages. In addition to correctly dealing with scheduling and synchronization of user messages, the implementation model should also handle concurrency issues such as fairness, deadlock, and application termination (Peterson and Silberschatz, 1985).

Deadlock in a concurrent system is a situation in which two or more processes are prevented from proceeding because each has allocated to it a resource required by the other. Deadlock in communication is not possible in *Enterprise* since the graph is essentially a call tree. Deadlock in allocating resources is handled by serializing the requests for resources, for example, requests for workers must be synchronized.

Fairness in operating systems is concerned with ensuring that a resource will eventually be allocated to the process requesting it, and that no process should wait permanently for a resource. The issue of fairness occurs when calling assets compete for workers to serve their calls, and also when workers compete to serve an incoming call. Fairness is also a concern when assigning processes to machines. Any process waiting for a machine should be able to run on an idle machine eventually.

Application termination deals with removing processes when they are no longer needed. Every worker process in *Enterprise* runs in an infinite loop in which it reacts to incoming call requests and executes the entry procedure. Unless the application finishes or detects an exception condition, a worker cannot be removed since a call request may arrive at any time. Therefore, the fact that the worker has been idle for an extended period is not a sufficient criterion for termination. A worker may end up losing some incoming calls travelling on their way along the network. In sequential programs, usually, when the main program finishes, the entire application is done. On the contrary, in concurrent execution, this is usually not the case. Consider a simple example of a line. The last process in the line may not even have started to process its final set of call parameters when the main program, which is the first process in line, finishes. All these issues must be considered when designing the termination algorithm.

To deal with the synchronization and concurrency issues described above, the managers class is introduced. This class of objects monitors and allocates the usage of various resources. These objects also ensure the proper termination of processes. Three behaviors are common for every member of this class: it handles requests for resources, it redistributes (remove or create) resources based on their demand and supply, and it detects when processes are finished and releases the resources that it monitors.

In this section, subclasses of the class managers are also presented. Since different resources may require a different way of handling, and different resources may have different utilization characteristics, it would be better to factor out the responsibility of managing these resources and have a separate manager managing each kind of resources. This management style is commonly found in real organizations, for example, there are usually managers for personnel, for factory lines, and so on. A manager class is named by the type of asset it monitors, as in real organization. The resources that exists in the execution environment are assets and machines. Factoring out the responsibility at design stage does not imply that these managers must each be run as a separate process. These issues will be considered at implementation time in the next chapter.

6.6.1. Asset Managers

The discussion on asset communication in the previous sections has identified the need for an asset manager to monitor an asset to guarantee that the use of it will be mutually exclusive. There should be logically an asset manager for every asset. The asset manager shares the name with the asset it manages, so that to the external objects, the asset manager represents the managed assets. The asset manager accepts requests from workers to register themselves as idle workers. The asset manager maintains a queue of idle workers as its private data structure to guarantee fairness in the selection of workers to perform calls. This ensures that a call is made to an idle worker instead of one that is busy. When a worker is done with a call, it registers with the manager so that it will be used again.

The asset manager can be migrated while keeping its private data in a consistent state. When the asset manager detects that a new instance of itself is created, it transfers its data state to the new instance and terminates itself. The individual manager is the asset itself. When a client makes a call to a pool or contract asset, the call is actually intercepted by the manager and then forwarded to an idle worker without the client's awareness.

In addition to mediating calls between clients and workers, the asset manager collaborates with the machine manager for ensuring better resource utilization. The machine manager monitors machine availability but has little knowledge of the calls among the assets. The asset managers, on the other hand, have no concern for the machines which the assets are run on. The asset manager handles requests from the machine manager to dismiss workers on busy machines. The asset manager can fire a worker when it feels that there is a surplus of workers. This will release the machine occupied by the fired worker, and the worker being laid off is communicated to the machine manager, who then updates the machine resource condition. The asset manager collaborates with the machine manager to ensure proper termination of the processes. This section will be focused on the manger's function as a synchronization monitor. Its other role as a "terminator" will be discussed in Section 6.7.

The class of asset managers serves as the superclass to define operations shared by its subclasses Pool Manager, Contract Manager and Individual Manager. Individual manager is an abstract class since the manager is the asset itself.

6.6.1.1. Pool Manager

As a subclass of the asset manager, a pool manager inherits all the behavior of an asset manager. Moreover, when a pool manager has all its workers on its idle queue for some time, it reports to the machine manger that its managed pool is inactive. The machine manager can therefore make better use of the inactive pool's machines. Once a client starts to access the pool, the pool manager reports to the machine manager that the pool is active.

There may be problems associated with the heuristic used to determine if a pool is inactive. For some application, the computation can call a pool, perform some local computation, then call the pool again. The pool may then be kept switching between active and inactive states. It all depends on the magic number for the time which a worker idles to be considered as inactive. In fact, every CPU scheduler is faced with a similar problem and a worst case scenario can be constructed for every heuristic.

6.6.1.2. Contract Manager

A contract manager also supports all the operations for asset managers. In addition, when a contract manager has more than one worker on its idle queue, it returns the extra worker to the machine manager. The machine manager will then remove the worker and releases the machine to the free pool of machines. When a contract manager runs out of idle workers to satisfy the call, it makes a request to the machine manager, which spawns a new worker if a free machine is available. The request will be repeated until there are sufficient workers to handle the call. The machine manager will try to meet the demand, perhaps not immediately, depending on the resource availability at the time the request is made.

6.6.1.3. Individual Manager

In the restricted model, individual assets communicate with each other directly. There is only one resource and so no manager is required to mediate the use of it. However, for completeness, so that logically every asset has its manager, the class individual is still defined. However, an individual manager is the individual itself. It satisfies all the operations required by the class asset manager.

6.6.2. Machine Manager

In addition to asset resources, the performance of the environment also depends on the effective management of machine resources. Since the tasks involved in assigning workers to clients is different from that of binding machines to processes, a new class is created to organize machines and processes. This is the only class which contains the state of the binding between machines and processes. The other objects in the environment can be freed from concerning about the location of the processes.

The machine manager allocates machine resources. This involves initial placement of assets on machines before the execution, and load balancing after the execution begins. Load balancing in this system involves first, the shrinking and expanding of contract assets on the machines based on their demand and availability, and second, process migration of processes, both assets and managers from busy machines to idle ones.

The information obtained from the machine secretary is used to classify machines on the network as being busy, free or used. A machine which is busy should not be assigned any asset. The term busy is overloaded with meanings here. It means that assets are unwelcome on that machine, due to real resource contentions or political reasons. A machine without any asset is free, and a machine with an asset already running on it is used. The machine which the user starts his main module on (presumably the user's terminal) is always marked as usable so that this can serve as the last resort to put processes on and avoid deadlock in the allocation algorithm.

The allocation begins by assigning assets and managers to free machines. The assignment is made by a simple depth first traversal of the call tree. If no free machines remain, machines which are already used are awarded to assets in a round-robin manner. Running several assets on a single machine achieves no parallelism but resource contention. Placing multiple assets on one machine is only a temporary measure to get the program in execution. When resources become available, process migration will be done to resolve the contention. Alternatively, the user has the option to wait until sufficient machine is available to run each process on a different machine. The user may also choose to not execute the application when there is not sufficient resource. The machine manager will inform the user of resource shortage and asks the user to decide. Every worker of the pool asset will be placed on a different machine if possible. Contract assets are allocated on a demand basis and initially only one instance of every contract asset is placed on the machine.

After the binding of processes to machines is completed, the machine manager launches the assets in the reverse order of the call tree. The assets at the leaves of the call tree are launched first. This order satisfies the call dependency relations and thus minimizes the polling time for an asset to establish connection with the called modules.

The machine manager also supports process migration when machine load changes are reported by the machine secretary. The secretary sends to the machine manager a message containing a list of machines with status of busy or free. If the process on the busy machine is of contract type and it is not the only instance of contract remaining, the contract is simply removed from the machine. If the asset is not a contract, the asset is migrated on to another machine. Migration is done by launching a new instance of the asset on another machine and then removing the original asset. If the user forbids process migration, the asset is left intact. The selection of migration destination is made according to a priority scheme outlined below.

When a machine becomes busy, a new machine must be found to migrate the asset on to. A free machine is attempted to begin with. If there is none, a machine is obtained from the contract. An idle contract is requested from the contract manager. The idle contract can be removed and the migrating asset can then be moved on to the free machine. If no free machine can be obtained by removing a contract process, a machine which has already been loaded with asset(s) is found. Initially, machines with inactive assets are attempted. If there are no inactive assets, the process is migrated onto a machine with active assets. The machine which is then overloaded with assets is remembered. When other machines become available, process migration will be done to resolve the overloading, first on machines with active assets, then machines with inactive assets.

When there are free machines, machines with assets that are active are migrated from the overloaded machines to the free machines. This ensures that resource contention will be resolved as soon as possible. Next, assets that are not active, but crowded on a single machine are moved. After this if there are still free machines, they are queued.

The machine manager is responsible for collecting machines released by a contract manager. When a contract manager has more than one idle worker, the manager terminates the extra worker and returns the machine to the machine manager. Also, it is responsible for launching new machines for contract assets when requested by the contract manager. If no idle machine is available, the request is queued and will be satisfied when a machine is available. The request for contract is at a lower priority than requests for machines to resolve machine contention among active assets.

The machine manager is responsible for marking an asset as being active or inactive when informed by the asset manager. The machine manager uses this information to place assets on machines when putting every asset on a separate machine is not possible.

The machine manager is also responsible for terminating or aborting all created processes when requested. When the machine manager terminates the processes, it will take into account that some of the processes have not yet finished even though the processes at higher level in the call graph have. The termination proceeds in the order of the call graph and will wait for work to finish properly at each level before proceeding to the next level. When aborting the processes, waiting is not required and all the processes are terminated right away.

Finally, the machine manager traps faults in any of the processes it launched and cleans up any residue processes left on the network.

6.6.3. Execution Manager

An interfacing object is required to serve as the mediator between the other parts of the executive and the external objects requesting services from the executive. The execution manager interfaces the kernel to the other components in the overall *Enterprise* architecture, such as the code librarian and application manager. Given an *Enterprise* graph, it initiates requests to the machine manager to place assets, except the main module on machines. When the placement is completed, the execution manager is responsible for launching the main module. It also interfaces requests from the application manager to change execution settings for debugging and process migration.

The next major function of the execution manager is to trap interrupt and program faults and cleans up any residue processes left on the network. In *FrameWorks* the process of application termination is initiated by the main module. When the main module completes the execution, it sends notices of termination to all its called modules. The termination procedure then propagates to all the processes along the call graph. This method only works if all the modules terminate normally. If the module stops prematurely, by user interrupt or due to an error in one of the modules, there may be runaway processes. Such a problem is avoided in *Enterprise* .

6.7. Termination

In addition to correctly dealing with scheduling and synchronization of user messages and machines, the implementation model should also handle concurrency issues such as application termination. Every *Enterprise* process, with the exception of the main process, runs in a loop to serve incoming calls placed on them until exit is explicitly called or an exception occurs. When the main program is done the algorithm *finish* is run. When an exception occurs to any of the processes the algorithm *abort* is run instead. Both algorithms ensure that there will be no residue processes of the *Enterprise* application on the network after the application aborts or finishes. These algorithms force processes along the call tree to quit one by one.

6.7.1. Abort

The *abort* algorithm should be run when the user interrupts the program, or when the program has an execution exception. The execution manager traps interrupts sent to the main module and sends to the machine manager the abort request they are received. The machine manager sends an abort request to all the asset managers in the application and calls exit to terminate itself. Upon receipt of the abort request, each asset manager sends to all its workers an abort request and then calls exit to terminate itself. Every worker, when it receives the abort request, calls exit to terminate itself. The abort request propagates along the hierarchy of the call tree, and ends in the individual workers which is always at the leaves of the tree. All the processes should be terminated at that point. Abort requests are all asynchronous and the requester will not wait for acknowledgement. Since the recipient calls exit immediately, work in progress and messages outstanding will be lost.

The machine manager detects exceptions in any process it launched and execute the abort routine when they occur. When any of the processes is terminated by sources other than the machine manger, it indicates that an exception has occurred in the terminated process. All the processes will be terminated in this case. The machine manager does terminate processes as it migrates them. It also terminates the processes when the application finishes normally, so the manager must rule out these cases and must not run the *abort* algorithm. In UNIX, processes which are done should be properly acknowledged by the machine manager to avoid the creation of a large number of *zombie* processes. These zombie process, which are dead, would still occupy space in the UNIX operating system's process entry table and may hinder the system's performance.

6.7.2. Finish

The algorithm *finish* is run when the main module, which the execution manager waits for, is done. The execution manager calls machine manager which then executes its own routine to finish up other processes and itself. The machine manager issues a request to finish which awaits reply to each of the asset managers in the hierarchical order of the call graph. The request blocks until the current asset manager is finished before the next manager is asked to terminate. This is the major difference between algorithm *finish* and *abort.* In many situations (for example, in line assets), after the main module has finished, the other modules may still be in progress and even have messages outstanding. This is especially true for assets at the leaves of the call tree. Unless the modules at the higher levels (closer to the root) of the call tree have completed all the work and messages, terminating the modules at the lower levels (closer to the leaves) may result in loss of messages. The asset manager upon receipt of the request to terminate, checks if there are outstanding messages on the system's message buffer or in its private queue. If there is none, the asset flushes its outgoing message buffer, sends requests to terminate all its workers and waits for the workers to reply. Since an asynchronous broadcast method is always used in implementing *Enterprise* module calls, it is necessary that messages will still reach their destinations when an *Enterprise* process terminates. When the process dies before its messages reach the destination, the messages may be discarded by the system. It is necessary to ensure that every message will be delivered to each of its destinations even if the process sending or receiving the message fails immediately afterwards. After it collects all the replies, it informs the machine manager that the particular asset is done and terminates itself. If there are outstanding messages, the request to terminate is delayed and will be served only after all the messages have been handled. Moreover, since the worker may be executing the entry procedure and cannot be killed when the request to terminate itself is received. The worker indicates that it is in the entry procedure by setting the flag busy before calling the entry procedure, and clearing it afterwards. The entry procedure is considered as a critical section that cannot be interrupted.

6.8. Conclusions

The discussion of the termination issues leads the chapter to an end. In this chapter, the design of the *Enterprise* executive was described. An object-oriented design methodology was used and this chapter shows how the architecture is evolved using this model. The architecture of the executive component of the system were described in detail. Several design alternatives were considered and some concurrency issues were discussed. The approaches to resolve these issues were presented. The following chapter describes the implementation of the *Enterprise* system based on the design presented in this chapter. Some implementation trade-offs will be discussed.

Chapter 7

Implementation of the Executive

7.1. Introduction

Although an object-oriented approach is used in designing the kernel, the implementation was written in C and uses the ISIS libraries. This combination was chosen for easy integration with the rest of *Enterprise*. A large portion of the system has been implemented on top of existing C programs, for example, the *Enterprise* compiler is implemented by modifying the GNU gcc compiler which is written in C (Chan, 1992).

Although the use of an object-oriented language is not strictly necessary to accomplish object-orient design, it is better to have object-oriented support from the language. ISIS provides the support for structuring the application using the object-oriented approach (Birman, Joseph, Raeuchle and Abbadi, 1985). An ISIS process is structured as a collection of message entries, which is equivalent to methods in object-oriented programs. Every message entry has an associated selector and an entry is invoked by messages sent to it. A new task for the entry is created on each incoming message and tasks run concurrently. Moreover, ISIS provides a process grouping mechanism which allows a group of process to be manipulated as if they were a single entity (Birman, Cooper and Gleeson, 1991). This modularity construct provides the necessary support for grouping instances of processes into classes.

7.2. Structure of an ISIS Application

ISIS applications are usually structured as a collection of message triggered tasks, which handle messages in a callback style. For each message sent to an entry, an ISIS task is created dynamically to execute the function specified in `isis_entry(function)` and the pointer to the message is passed as a parameter to the function. A task is composed of a memory area where a C subroutine call stack can be managed and a descriptive structure in which register variables used by the task can be saved when the task is blocked. ISIS recommends the call-back style of programming in which task creation occurs on each message. Spawning a new task for every message is simple, avoids risk of deadlock, and achieves maximum concurrency. The mechanism corresponds nicely with some object-oriented paradigm for message handling, for example, *Actors* (Agha, 1986) since a new instance of itself is created on every incoming message. An ISIS program will be message driven and the user does not have to worry about polling or blocking for messages.

Since the UNIX kernel does not support the task mechanism, running tasks without system support imposes several limitations:

1. A UNIX stack can grow without limit, but ISIS tasks are subject to a stack size limitation. A normal UNIX program can recur or dynamically allocate memory effectively without limit. Tasks run on dynamically allocated stack space obtained from `malloc` and not the normal system stack area of a UNIX program. Local variables declared within a procedure or block, arguments, and register variables are saved on the stack in each call. The current stack limit is 16 Kbytes. Locally declared large data structures can exceed this stack limit, as can deep recursion. Although one can set the stack limit by calling `isis_entry_stacksize(entry, size)` to specify a size to use with the message entry, in *Enterprise*, it would be hard to predict the user's program stack usage behavior at compile time to insert the appropriate call.

2. Since a task is external to the operating system, its implementation costs both time and memory space. Task creation incurs an overhead of about 1 ms on a typical UNIX system (ISIS, 1992). ISIS tasks consume a lot of memory, even when they are blocked.

3. The achieved concurrency would not enhance performance on a uniprocessor system, but promote resource contention. Moreover, for certain applications, it is required that requests be handled one at a time to ensure deterministic execution. For example, in *Enterprise*, an individual is required to behave as a single thread sequential program.

Since ISIS tasks have considerable overhead and some limitations, tasks are avoided in implementing *Enterprise* applications. Although the internal implementation does not use the task mechanism, a process can still be viewed from the external as if it were using the task mechanism. Messages are still labelled by an integer selector which directs it to the appropriate entry point in the program. The program must explicitly receive the messages arriving at the entry point by calling an ISIS receive message system function. The entry procedure is not subject to the stack limit, incurs a much smaller overhead, and the next request waits until the current one is done. Unfortunately, it will still be subject to the limitation of a fixed message buffer. There is simply no apparent solution to this problem, as experienced also with the UNIX `socket` facilities itself (Li et al., 1992). The program will be an inter-mix of conventional single thread and task style. ISIS tasks are still used in handling special messages for terminating or aborting the program, in similar ways as signal handlers are used in normal programs. In such cases the stack limit and overhead are not concerned since these tasks are executed only once.

7.3. ISIS Broadcast Primitives

Messages in ISIS are sent by broadcast calls. It is similar to the semantics used in Remote Procedure Calls except that the message can be directed to multiple destinations in a single call (Birrell and Nelson, 1984). The ISIS broadcast primitive `fbcast` is the least costly broadcast primitive in ISIS. It provides FIFO ordering on a point-to-point basis and it is reliable. The term reliable means that either all the destinations receive the message or none do, even if the sender fails. For the *Enterprise* system, the calls are usually sent point-to-point instead of multicast, which is only used when multiple processes all execute redundant computations to achieve fault tolerance. Therefore, the ordering of message delivery in the group is not important and `fbcast` is used because of its low cost.

Normally ISIS sends messages indirectly via a program called *protos*. ISIS also supports another protocol known as the BYPASS communication, in which messages are sent directly to their destination directly without going through *protos*. BYBASS communication is used when the following properties are satisfied:

1. The communication is to be sent to a single destination
2. The destination is a group to which the caller belongs or the caller is a client of.
3. The broadcast protocol must be one of mbcast, fbcast, cbcast or abcast[1].

Broadcasts that do not satisfy the above criteria will be sent using the slower protocols.

7.4. Limitations Imposed by ISIS

The version of ISIS used is v2.2, which is effectively identical to the commercial version v3.0. There are several effects on any application that uses ISIS.

1. Users would be required to purchase the ISIS software before they can enjoy the benefit *Enterprise* offers.

2. ISIS restricts the naming of user variables. Many ISIS global variable and structures use popular names, for example, message and address. These names are common in the user's code and become a problem when ISIS code is inserted into the user's

[1]The specify semantics of the broadcast protocol is beyond the scope of this thesis, and can be found in the ISIS documentation.

code. The naming conflict actually arise when the *Animation* application described in Chapter 4 is compiled.

3. Another limitation concerns the message buffer size of ISIS. ISIS may choke when messages are sent rapidly without waiting for acknowledgement, as in asynchronous broadcasts. When unreceived messages are queued up and the backlog of messages gets sufficiently large, ISIS will discard new messages and send the channel overflow signal and kill the program. For example, when the program is blocked for I/O, the program will stop reading messages and message overflow is possible. The limit of backlog limit is 32 Kbytes plus 8 Kbytes of data that can be in the channel. Currently there is no way to negotiate with ISIS over the threshold for message backlog.

4. ISIS processes are quite large. This causes considerable disruption when running the program on a distributed environment shared by a user community. Running the program on other users' machines may cause flushing of their file system cache contents. When they return, they will find that their favorite files have been removed from the cache to make room for the files of the *Enterprise* processes.

7.5. Implementation of Assets

As discussed in the previous chapter, an *Enterprise* program in execution is a set of communicating objects. Objects which execute a user module are known as assets based on the discussion in the previous chapter. Each asset is uniquely named by the module it runs. Asset running a module may have more than one instance, if the asset implements a parallel structure with multiple processes (for example, pools and contracts). Each asset must establish connection with all its communicating partners when they start up. It has to announce its availability on the network as well so that others can communicate with it. It also needs to know its execution state as being busy or idle so that it will not be terminated by its manager when it is busy handling a call. It may be running in debug mode, in which every message exchanged will be carbon copied to the debugging monitor. The mode of execution is recorded also in global variables. It is also necessary to declare methods and their associated selectors to handle messages for executing a module call and messages to terminate itself. The sequence of events that takes place when one process calls another is shown in Figure 7.1. The shaded blocks indicates the work performed by *Enterprise* , transparent to the user, when a call is made. The compiler inserts the necessary stubs for performing the message exchanges and the kernel manages the communication at runtime.

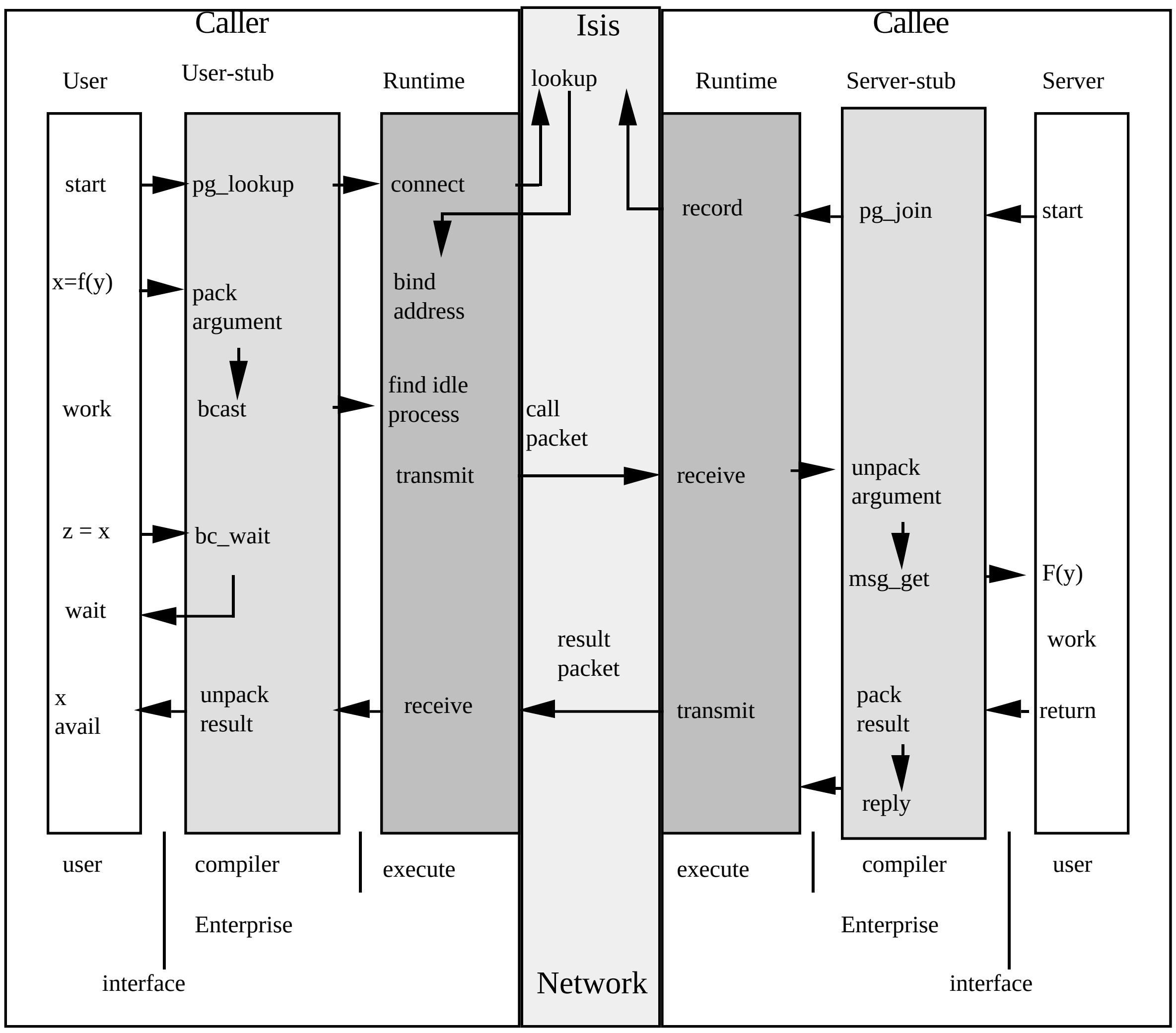


Figure 7.1. Sequence of events in an *Enterprise* RPC

7.5.1. Main Structure Implementation

All the asset processes in *Enterprise* have the following general structures:

1. The program calls `isis_remote_init()` to connect itself to the ISIS server.

2. It declares task entries by calling `isis_entry()` to handle termination and procedure call requests.

3. It registers itself to the asset group by calling `pg_join()`, which makes itself known on the network.

4. The program obtains the addresses of all its called assets by `pg_lookup()`.

5. The program ends the start up sequence by calling `isis_start_done()`.

6. The program then enters a loop to read messages and calls the entry procedure with the received messages. If no message is available, the program blocks until one is available.

7. At the end of the loop the program calls `isis_accept_events()` to give other ISIS tasks, such as the termination or abort procedure, a chance to run.

The code to perform the above is inserted as generic templates into the user's program and does not require the processing of the user's code. Instead, the exact code inserted depends on the kinds of asset that the module is coerced to. The user's code is kept as intact as possible, except at entry procedure declaration and entry procedure calls, so that the code will not be hard to read even after the communication code is inserted. Also, unless the user make changes to the program code, coercing an asset to a different kind requires only relinking the code template of the new asset. Recompilation of the user program is not necessary. Appendix C illustrates the generic template inserted for the modules in the *Animation* application. The program structure outlined above will be discussed in detail later in this chapter.

7.5.1.1. Establish Connection

The following ISIS routine must be called to initialize ISIS before calling any other ISIS routine. This call should be made before using any other ISIS code:

```
isis_remote_init(name_of_server, lport_nr, rport_nr, option);
```

This routine should be called in the main routine and the value `lport_nr` is the port number used by ISIS to talk to applications. The value `rport_nr` is the port number used by ISIS to talk to applications. The value is chosen by the system administrator when installing ISIS. If the number is given as 0, ISIS will obtain the value from the environment variable `ISISPORT` or look in the `/etc/services` file.

The ISIS routine `isis_entry(ABORT, abort, "abort")` defines that when a message is delivered to the entry point, `ABORT`, a new task of the specified routine, `abort`, will be created to handle the message. The last argument in the subroutine call is just a string for descriptive purposes. An ISIS process can have many entries and each one is given a unique entry number and messages are addressed to these entry numbers. As mentioned in the previous section, ISIS tasks are only used in handling special messages for terminating or aborting the program.

Procedure call messages addressed to the entry procedure are not handled by the task mechanism for better performance. If the routine is specified as `MSG_ENQUEUE` in `isis_entry`, instead of spawning a new task on every incoming message, messages to the corresponding entry point are queued up and must be individually received.

The ISIS routine `pg_join("B",0)` registers the process as a member of the named asset group. The first argument must be the name of the process group to join. The last argument must be a 0 which marks the end of argument list. In between the two arguments, some optional keywords may be specified, for example, to perform process logging for fault tolerance. The routine `pg_join` returns the address of the group. All *Enterprise* assets in the application except the one running the main module would be used by some other modules and must register itself via `pg_join`. When inserting communication code in the application, an asset can be recognized as the main module from the call tree. The main module is at the root of the call tree.

If an asset makes calls to other assets, it must obtain the addresses of those assets using the ISIS routine `pg_lookup`. This routine accepts an asset name as its parameter, for example, `pg_lookup(asset)`, and returns the address of the process group. ISIS supports naming transparency and the address of a process group can be obtained simply by its name, regardless of its location in the networked environment. To speed up message delivery via the ISIS BYPASS mode, the asset can be registered as clients to all the assets it calls. When code is attached to a module, all the assets called by an asset can be determined from the call tree and appropriate `pg_lookup` routines can be inserted. Since all the services are potentially used by every asset (except services), for simplicity, calls to look up all the services addresses is inserted in the start up sequence of every asset (except services).

ISIS inhibits the delivery of messages from other processes until `isis_start_done` is called. This ensures all the necessary initializations are done before the process responds to any incoming message. The routine `isis_start_done` informs ISIS that the start up sequence is completed and messages can be sent to the process.

7.5.1.2. Receiving Call Messages

After the start up sequence is done and `isis_start_done` has been called, the program runs in a loop to receive messages addressing to run the entry procedure, to terminate or to abort the program. Messages to the entry procedure can be received using the call `msg_p = msg_rcv(entry)`. The `msg_rcv` call will block until a message is available. The variable `msg_p` will contain the address of the incoming message. The argument `entry` is an integer serving as the message selector, such that messages can be sent to this location and retrieved using the selector key. It should be declared as a message pointer: `message *msg_p`. Since messages addressing the finish and abort routines will be run as ISIS tasks, these routines will be invoked even when the program is blocked at the `msg_rcv` call. They will be discussed in Section 7.5.1.4.

After the message is received, the process sets the busy flag and call the entry procedure passing the message as parameter. When the entry procedure returns, the process clears the busy flag. The busy flag signifies that the entry procedure is in execution when the process is interrupted by finish messages. If the busy flag is set, the process cannot be terminated.

If the program does not block in ISIS calls, other ISIS events may not get a chance to run. At the end of the loop, `isis_accept_events(ISIS_ASYNC)` is called to read and deliver any pending messages to other tasks. The control will be returned to the caller after the resulting tasks had a chance to run. The flag `ISIS_ASYNC` tells ISIS to run asynchronously and continue without blocking when there is no other task to run.

7.5.1.3. Synchronization

If an asset has multiple instances, as in pools or contracts, before a worker waits for incoming messages, it sends to its manager its address. Using the address, the manager can forward work to a worker. The ISIS variable `my_address` contains the address of the process. The usual way to address a specific process in a group of identical process, as suggested by ISIS, is by its *rank*. The oldest member in the group has rank 0, the second oldest rank 1 and so on. Each process in the asset group therefore has a unique rank number, which the asset manager can use to address work to a specific process. However, in the *Enterprise* environment, processes are created and destroyed dynamically. For a contract asset, the number of multiple processes changes dynamically, and thus their ranks. For a pool asset, which has a static number of processes, the ranks of the processes may also change with time due to process migration. There is no guarantee that the rank received by the manager is the most up-to-date information. When the rank information is on the way through the network, the actual rank may have already changed in the worker. Since in ISIS, an address of a process is unique and static with time, the following code is inserted to announce a process' identity before waiting for an incoming message:

```
fbcast(<module>,CHECK_IN,"%A[1]",&my_address,NREPLY);
```

7.5.1.4. Termination

ISIS task entries are used to implement the termination algorithms. The algorithms *abort* and *finish* are declared as an ISIS tasks using the following primitives:

```
isis_entry(FINISH, finish_handle, "finish_handle");
isis_entry(ABORT, abort_handle, "abort_handle");
```

The `abort` entry takes a message as parameter and calls `exit` immediately.

```
abort_handle(msg_p)
message *msg_p;
{
   msg_get(msg_p,"");
   exit();
}
```

The `finish` entry also takes a message as an argument but instead of quitting right the way, the request waits. The request should wait while the entry procedure is in execution or when there is message outstanding. There is a choice here as to whether the task should block waiting for the entry procedure to be done or the messages to be cleared, or the task should end and the request to terminate should be remembered. The latter approach is chosen because a task consumes a lot of memory even when it is blocked. Therefore two issues must be dealt with: first, the incoming request should be remembered so that a reply can later be sent to the requestor when the request is satisfied, and second, the number of outstanding messages must be known at any time.

The information pertaining to a sender resides only in the message and it is not possible to regenerate this information. Therefore, the message must be kept around for a reply to be sent to the correct requestor when the request is served. Normally, ISIS will automatically delete messages when a task is done. Each message has a reference count which is 1 by default. ISIS decrements the reference count to delete a message. When the reference count reaches 0, the storage of the message is reclaimed. To keep ISIS from actually removing the message, the routine `msg_increfcount` is called so that the reference count will not reach 0 when the entry returns.

The number of outstanding messages at an ISIS entry point can be obtained by calling `msg_ready(entry)`. A program should not be terminated when the number is non-zero and when the entry procedure is running. Since ISIS tasks are run concurrently, an explicit lock must be used to safeguard the entry procedure. The lock is implemented by global variables which are shared by all tasks. The variable `_e_busy` is set when program is in the entry procedure so that the procedure will not be interrupted.

When the required condition for termination is met, the program must flush its message buffer before quitting. This forces all messages that may be buffered in the ISIS system to be delivered to their destinations. The ISIS primitive `flush` is called to accomplish this.

This section illustrates the code required to implement different assets. The asset dependent code is inserted to the user's module, and is independent of the content of the user's module. The responsibility of the code includes establishing connections with other processes, accepting incoming messages and running the entry procedure, handling of termination, and ensuring fairness of use in assets of multiple processes. Since the above code, which implement the asset's parallelism, is independent of the user's code, it can be compiled as a separate library. Coercing a module to use a different asset usually requires only relinking the module to use a different library, unless the user code has been changed. As mentioned in Chapter 4, coercing an asset to a line or department will require change in the user's code, and so requires recompilation of the user's code in these cases. The next section describes the generation of code for making procedure calls which are dependent on the user's code.

7.5.2. Call Implementation

The following section presents the mechanism to make *Enterprise* calls and to extract the parameters from a received message. Parsing of user code is required to turn the entry procedure to a procedure which accepts a message as parameter and extracts from it, procedure call parameters. Processing of user code is also needed to transform entry procedure calls into ISIS broadcasts. The compiler is responsible for the transformations. This section gives the specification of its code generator component. The implementation is researched by Enoch Chan (Chan, 1992) and is done by modifying the GNU gcc compiler.

7.5.2.1. Gathering of Call Parameters

The parameters used in a module call are encapsulate in a message which is sent to the module being called using the ISIS broadcast primitive. ISIS ensures that appropriate conversions are made if the representation of a data type varies from one machine to another (for example, VAX integers are stored differently in memory than most other machines). The number and type of the parameters to be exchanged must be specified by a format string in a manner similar to the C formatted I/O functions such as `printf` and `scanf`. ISIS supports all the predefined data types in C. The following is a list of the predefined message format types (Birman, 1992):

```
%A   address structure pointer or vector
%B   bitvec (vector of 128 bits) pointer or vector
%c   char
%C   char pointer or vector
%d   long int  (4-byte)
%D   long int pointer or vector
%E   event_id structure pointer or vector
%f   single-precision (32-bit) float.
%F   float pointer or vector.
%g   double-precision (64-bit) float.
%G   double pointer or vector.
%h   short int  (2-byte)
%H   short pointer or vector
%l   long int  (4-byte); syn. for %d
%L   long pointer or vector, syn. for %D
%m   message pointer
%P   groupview structure pointer or vector
%s   character array  (null-terminated character string)
```

Upper-case format items denote a vector of the corresponding base type. In the case of a vector, a length argument specifying the number of elements is required immediately after the vector address. For example, the parameters used in a call or a procedure header with types defined below

```
int size;
float priceTable[10];
char stockType;
```

will be translated into a message with the following format string and arguments

```
"%d%F%c",size, priceTable, 10, stockType
```

In ISIS, user-defined structures are supported by creating a new format string. It is therefore necessary to extract from the parameter list of an entry procedure the names and types of the parameters and record them in a symbol table. If the data type is not defined, a call to specify a format string for the type must be included:

```
isis_define_type(typeCharacter, sizeof(struct Parameter),converter);
```

The argument `typeCharacter` is an ascii character used in format items referring to this type. It must not redefine the ISIS predefined formats (a b c d e f h l m p s). This type character should be recorded in the symbol table so that it will be reused when a message of this type is constructed in other modules. The converter routine is responsible for mapping a data item of the specified type from the byte format of a sending machine into the byte format of the receiving machine. However, the data items encapsulated in the user defined structure must be known so that they can be converted in turn. In our current implementation, no converter is specified. This limits the application to run on machines that share the same byte ordering.

Since the available formats are limited by the number of allowable formats, which is small, a casting mechanism is used instead to handle user defined structures. User structures are cast into C characters of equivalent sizes. In sending or receiving the structure, the cast `(char *)` is used. The resulting scheme is even simpler to implement. This scheme also suffers from the limitation that the application must be run on machines sharing the same byte ordering.

7.5.2.2. Making a Call

Module calls that return a result are called f-calls and modules calls that do not return a result are called p-calls, as discussed in Chapter 3. A p-call is translated into the following ISIS broadcast that does not expect any reply, which is always non-blocking:

```
fbcast(<address>, CALL, <parameter format>, <parameters>, NREPLY);
```

fbcast is an ISIS broadcast primitive, as mentioned in Section 7.3. The argument <address> specifies the address of the called worker. The argument CALL is a constant which selects that the message should invoke the call servicing method of the worker. A worker object has also methods for handling debugging and termination requests and the selector distinguishes the different methods. The arguments <parameter format> and <parameters> defines the format and call parameters to be gathered into the message, as discussed in the section above. The argument NREPLY is a constant of value 0 which specifies that no reply is expected from the call.

An f-call is translated into an ISIS broadcast that expects a reply. Non-blocking broadcast is specified with the 'f' option in the bcast call so that a task is created to run the broadcast asynchronously. The call <return value> = f(<parameters>) is translated to the followings:

```
int <return value>_token;
<return value>_token = fbcast_l('f', <address>, CALL,
                                 <parameter format>,
                                 <parameters>,
                                 REPLY,
                                 <return value format>,
                                 <address of return value>);
```

A token is required for each asynchronous broadcast so that the return value can be received later by a rendezvous using the token. The argument 'f' of the broadcast call specifies that the broadcast is to be forked off as a task. There is the additional arguments REPLY, return formats and return values. Otherwise, the call is similar to that used in a p-call. The argument REPLY has a value of 1, which specifies the broadcast should expect a reply. The return value is the address of the variable in which the reply can be put. The format string of return value is declared in similar way as that in call parameters.

When the variable holding the <return_value> is referenced, the following line is inserted just before the reference is made, to block and wait for the result to arrive:

```
bc_wait(<return value>_token);
```

If this is the first reference after the call, this may cause the program to block and wait until the call returns. If the result is ready, the program resumes immediately. The program also resumes immediately when the reference is not the first one. For parameters and return value declared as follows,

```
int a;            /* 'a' is an integer */
int b[10];        /* 'b' is an array of integers */
struct point x,y; /* 'x', 'y' are user-defined structures called 'point' */
```

the three C statements in a program:

```
/* 1. making a p-call */
g(a);
/* 2. making a f-call and expecting result to be received in 'x' */
x = h(b);
/* 3. accessing 'x' */
y = x;
```

will each result in the translation or insertion of a call to the ISIS library:

```
/* 1. ISIS RPC expecting no reply */
fbcast(g_pg,CALL,"%d",a,NREPLY);

/* 2. ISIS RPC expecting a reply which can be collected via a token */
x_token = fbcast_l("f",h_pg,CALL,"%D[10]",b,REPLY,"%C",&x,sizeof(point));

/* 3. Variable accessed, 'bcwait' inserted to wait for RPC to return */
bc_wait(x_token);
z = y;
```

It is of interest to note that in 2, the address of the receiving variable is given as the argument, and, the user-defined structure is received as an array of characters.

7.5.2.3. Extracting Call Parameters

The entry procedure header and the parameter declarations are translated into an entry which extracts the parameters from an ISIS message. A format string for the parameters is generated based on the data type of the parameters. The format string and parameters are stuffed into a receive call which is inserted into the user code. The entry procedure `<function>`:

```
<function>(<parameters>)
<parameter type declarations>;
{
  <local variables>
  ...
}
```

is translated into

```
<function>(msg_p)
message *msg_p;
{
   <parameter type declarations>;   /* parameters declared as local variables */
   msg_get(msg_p, <parameter format>, <address of parameters>);
   {
      <local variables>
     ...
   }
}
```

For example, a stub procedure declared as follows,

```
g(x,y)
```

```
int x;
struct y;
{
}
```

will be translated to:

```
g(msg_p);
message msg_p;
{
   /* ISIS call to extract variables from a message. */
   msg_get(msg_p,"%d%C",&x,&y,dummy);
}
```

The addresses of the parameters are given as arguments, and, the structure is extracted as an array of characters. The `dummy` argument is a place holder. This argument was originally used to put the size of the array, which is not of concern here. Array parameters with dynamic size will not occur in *Enterprise* because pointers are not allowed as parameters.

After the call has been received and the entry procedure executed, a result may be returned. An ISIS reply call will be inserted before any return statement in the entry procedure:

```
return(<return value>);
```

will be translated into the statements

```
reply_l("f",msg_p, <return value format>, <return value>);
return;
```

which sends the return value to the caller using the fbcast mechanism (denoted by "f"). For example, if the return value is declared as follows,

```
int a;            /* 'a' is an integer */
```

the return statement:

```
return a;
```

will be translated to:

```
reply_l("f",msg_p,"%d",a);
return;
```

7.6. Implementation of Managers

Besides assets, the other class of objects that exists in the run time environment are managers. Initially, the manager was implemented as a single process encompassing all the responsibilities of the classes asset managers, machine manager, and execution manager defined in the previous chapter. This was the approach used in implementing the general model in

FrameWorks. There were two manager process, the *Execution Time Monitor (ETM)* and the *Butler* (Nichols, 1987). The *Butler* maintains a database that contains information on the load-status of machines on the network, which is similar to the UNIX *ruptime* program. The *ETM* oversees the execution of a *FrameWorks* application and performs many functions:

1) It carries out the initial hand-shake procedure to ascertain that all the processes in the application have started properly.

2) It coordinates the execution of calls and returns between clients and workers.

3) It executes the termination procedure of the entire application.

4) It may collect run time statistics and event history of the application which can be used in post-execution analysis.

The *ETM* is therefore the main control center of a *FrameWorks* application. The initial implementation for *Enterprise* followed a similar direction. However, it was soon realized that the single process which carries out all the different responsibilities and acts as a communication center soon became a bottleneck of the system. Collecting load averages and launching processes on machines usually requires substantial waiting time. In the mean time, a large number of call messages may be queued up in the system waiting to be routed to their appropriate destinations. Therefore, in the subsequent versions of the *Enterprise* executive, the responsibilities of the manager class are split amongst different processes. In the next section, the implementation of the asset manager, which is concerned with routing a call to an available worker process is discussed.

7.6.1. Implementation of Asset Managers

In assigning the work for an asset manager, there are two choices. The asset manager can route all the call messages from different modules in the application to their respective destinations. Alternatively, it can coordinate only the calls to a single module, having an asset manager to manage each module. The latter would have the advantage of enhanced transparency, since the asset manager can then represent a worker. A client is not concerned with whether it is calling a worker of multiple instances or a single worker. Moreover, this can avoid a potential communication bottleneck when several clients are calling different modules, since the calls will be routed through different asset managers. This may not mean substantial improvement in communication if the application is to be run on the *Ethernet*, since all the messages would still be jamming through a single bus. However, this approach exhibits more parallelism and a call to a module would not have to wait when the manager is busy coordinating the call of another module. Although the asset managers are managing different workers, they can be implemented using the same program and the binding of the worker to be managed can be done at run time. In terms of object-oriented approaches, this is known as different instantiation of the same class method. In the current implementation, a separate asset manager is launched at run-time to manage every asset (except individual, whose incoming calls are not managed because there is only one process to handle the calls).

The main responsibility of the asset manager is to forward calls from a client to an idle worker. When implementing the asset manager, there are two choices. A call request could block, waiting for an idle worker to declare itself, or it could be saved somewhere reasonable when there is no idle worker. The latter would be a more efficient solution because tasks use up a lot of memory. If no idle worker is available, the message is queued up. The message is prevented from being deleted by the ISIS system using the routine `msg_increfcount(msg_p)`. When a worker is available, it sends a message to the manager registering itself. On receipt of this message, the asset manager removes the first message in the queue and sends it to the worker using the ISIS `forward()` routine. This ISIS primitive will send the message to the worker while using the last sender as the sender of the message, so that the worker can reply directly to the original sender. If there is no message waiting in the queue when a worker registers itself, the worker is appended to the queue of idle workers. When the message has been forwarded to the worker, the message should be explicitly removed by calling `msg_delete()`. Otherwise, the accumulation of useless messages can result in a memory leak, in which the program gradually becomes more and more bloated and creaks to a halt eventually.

7.6.2. Implementation of Machine Manager

As mentioned in Chapter 6, the machine manager has three main responsibilities: to establish the necessary connection, to put the processes on machines, and to monitor the load condition in the environment. To carry out these responsibilities, the machine manager maintains the necessary data structures to record the dynamic binding of processes to machines. It updates the data structures when the binding changes. To ensure fairness in the allocation of machines to processes, several queues are maintained with each recording processes with different priorities. The queues also keep track of the machines which are free, used, busy and overloaded. To detect load changes, periodically, the resource secretary informs the machine manager of the load averages. The UNIX `rsh` command is used to put processes on remote machines in the environment.

7.6.2.1. Process-Machine Binding

The machine manager maintains the binding recording which machines each *Enterprise* asset runs on. Every *Enterprise* asset is implemented as a unique ISIS process group with the asset name being the group name. *Enterprise* assets (for instance, pool) may have multiple processes at run time. A group running the pool or contract asset thus has more than one process in the group. When a machine becomes busy, processes on the machine must be removed. The machine manager finds out what asset each process on the machine belongs to and requests the asset manager to have the particular group member removed. At any moment, arbitrary processes of an asset may be removed to migrate the processes or cut back a contract. Initially, the entire mapping of machines and processes is stored in a dynamic hash table as shown in Figure 7.2.

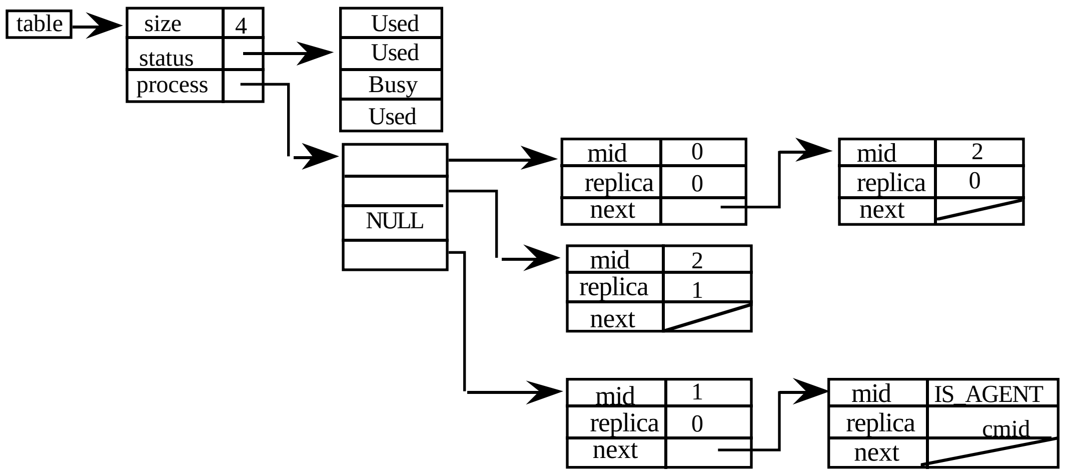


Figure 7.2. Preliminary Machine Table Structure

The processes allocated to the machines are put in the buckets indexed by the machine id's. The hash table data structure supports the requirement of binding more than one record with a hash index, and can handle the situation when more than one process is moved on to a machine. However, this table soon proves insufficient because updating a process on the mapping affects not only the machine, but the composition of the asset in some cases. The processes in a group running an asset are always ranked in their order of creation. For managers to address a call to the correct worker, up-to-date rank information must be maintained. Also, removing a process from the network not only subtracts a process from the machines, but also decreases the number of members of an asset. Initially, this information is scattered in different tables with some duplicated information found in the asset table and the machine table. The asset table is shown in Figure 7.3. The mapping structure is not only costly to update at run time, but is hard to maintain and understand since it does not correspond well with the real model of the operation. Therefore, the structure is redefined according to the operation required.

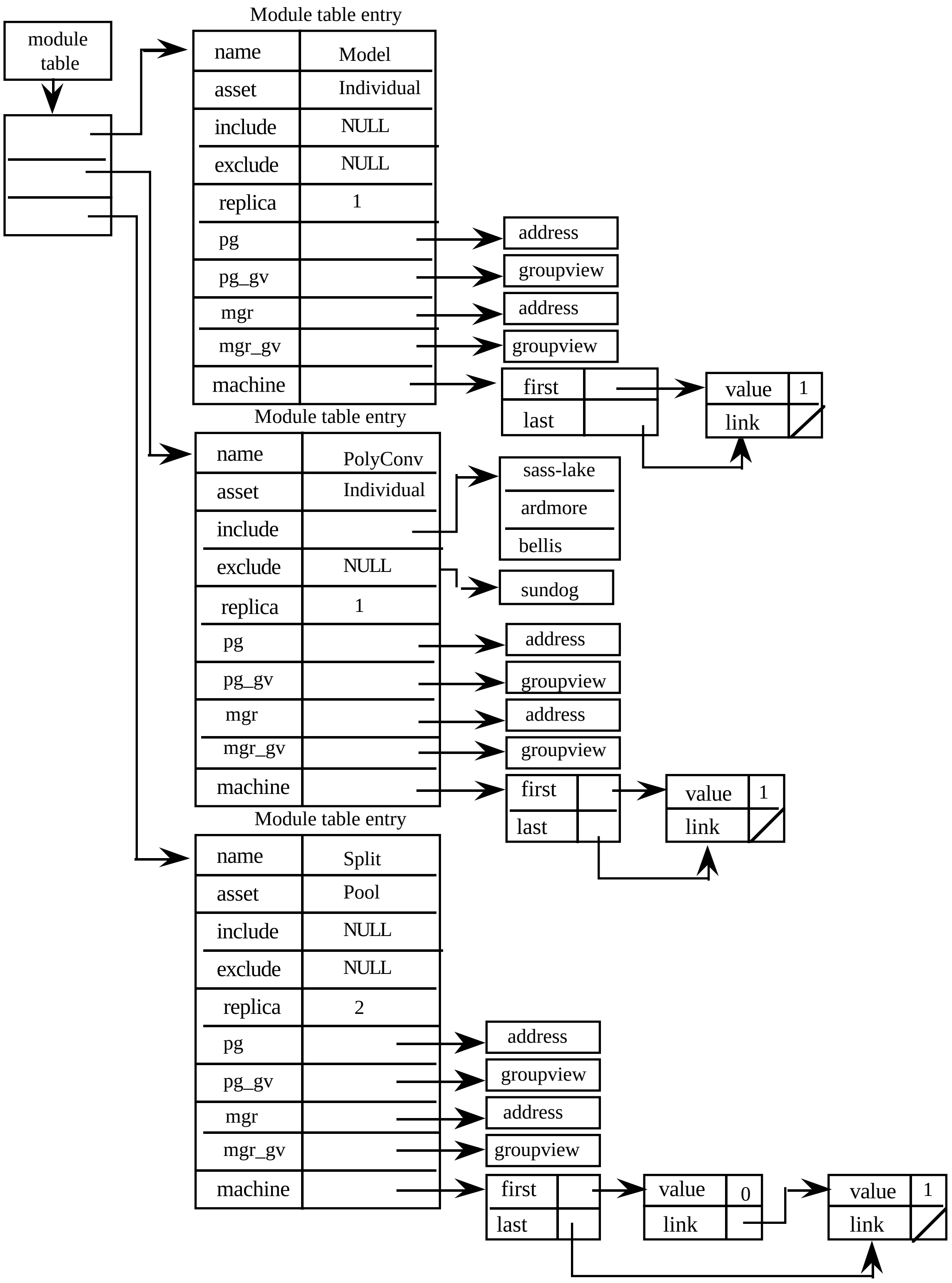


Figure 7.3. Preliminary Asset Table Structure

A doubly-linked list is used to organize processes in an asset. This structure allows for the convenient removal of arbitrary process and recalculation of the id of the other processes. The id of a process is not stored but computed by traversing the list. The id only needs to be computed or updated when a process is removed. Storing or computing the id does not represent any trade-off in computation cost. The latter costs less in terms of storage. The linked-lists of processes in different assets are organized as an array of linked-lists. Consistent with the ISIS convention, processes are organized to have the most recent process bearing the largest id. Although processes can be removed in arbitrary order, processes are always added to the end of the linked-list. Processes to run the manager of an asset are also organized as doubly linked-lists.

From time to time, arbitrary processes on a machine may be removed. The request to remove the process may come from two sources. A machine manager may remove a process from the machine when the load on it becomes heavy. A contract manager may request to have its surplus workers removed. These requests may involve updates of arbitrary workers. Therefore, a doubly-linked list is also used to organize processes on a machine.

When adding or removing a process of an asset on a machine, the process needs to be examined with regard to both the machine and the asset. Each process record should therefore be simultaneously an element of one of the machine list and one of the asset list. The resulting data structure is an orthogonal list. Each process record contains two sets of pointers, machine-links and asset-links, by which the orthogonal doubly-linked list are formed. This structure, which is the principle data structure maintained by machine manager, implements the allocation map of processes to machines. Each process contains, in addition to the links to the next process, a status flag to mark the process as active or inactive and a process id. When an asset manager (having all its member on the idle queue) considers the asset it manages as inactive, it informs the machine manager and the flag is set. When the machine manager launches a process, its process id is recorded. Whenever a `SIGCHLD` signal is received, the process id returned from *wait* is used to determine if the process has been abnormally terminated. Processes terminated normally by the machine manager would have their record removed and their process id would not be found. Child processes run by the machine manager are waited to avoid creating a large number of zombie processes. Figure 7.4 shows the process node structure of the following declaration:

```
struct struct process_node {
   int status, pid;
   struct process_node  *machine_next,*machine_prev, *asset_next,*asset_next;
}
```

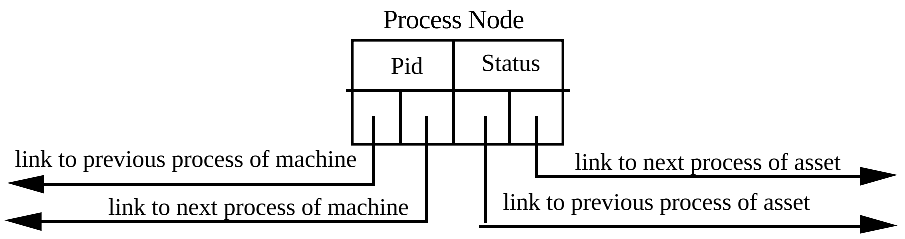

Process Node
Pid
Status
link to previous process of machine
link to next process of machine
link to next process of asset
link to previous process of asset

Figure 7.4. A Process Node Structure

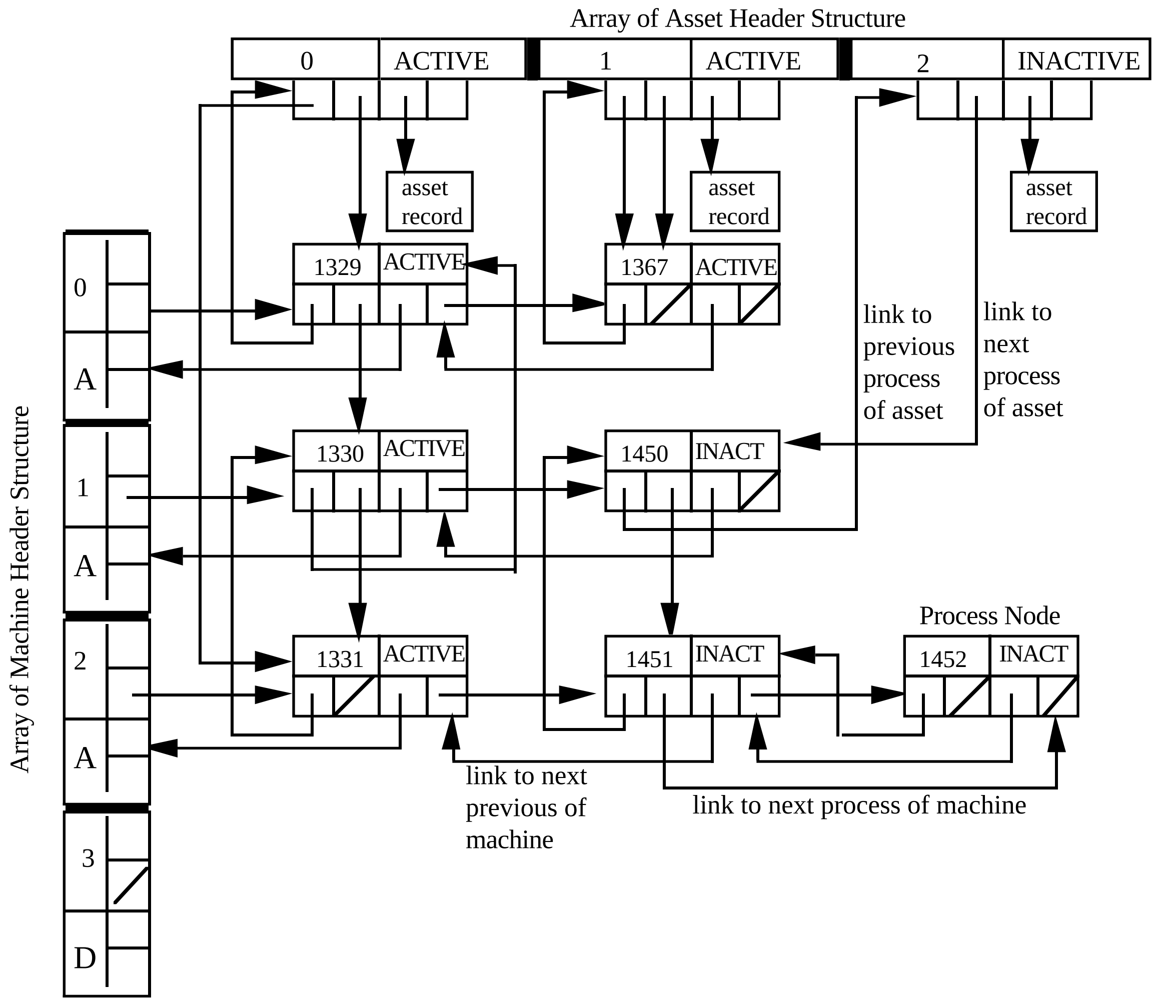

Array of Asset Header Structure
0
ACTIVE
1
ACTIVE
2
INACTIVE
asset record
asset record
asset record
Array of Machine Header Structure
0
A
1
A
2
A
3
D
1329
ACTIVE
1367
ACTIVE
1330
ACTIVE
1450
INACT
1331
ACTIVE
1451
INACT
1452
INACT
Process Node
link to previous process of asset
link to next process of asset
link to next previous of machine
link to next process of machine

Figure 7.5. Process-Machine mapping structure

Since doubly-linked lists are used in the structure, the header records in the lists are of same type as the process node. Figure 7.5 shows an example of the data structure configuration with several process node connected together as process are added to machines and assets.

An asset list header is different from a process node in that the machine links and the process id field are unused. The field for process id is used to hold the asset id, which is the index of the asset list header in the entire array of asset list headers. Assets in the array are arranged in the order of the call graph structure. Since service assets may be called by any other asset, they are always placed at the end after all other assets. The *link to the next processes of machine* field is used to reference the last item in the process list, so that processes can always be added to the end of the list. The *link to the previous process of machine* field points to an asset record which has a structure shown in Figure 7.6. The asset record records many useful attributes and characteristics of the asset, such as its name, kind, machine preferences, and number of members.

```
struct asset_record {
      char *name;
      asset_type kind;
      int nr_replica;
      QUEUE include, exclude;
   }
```

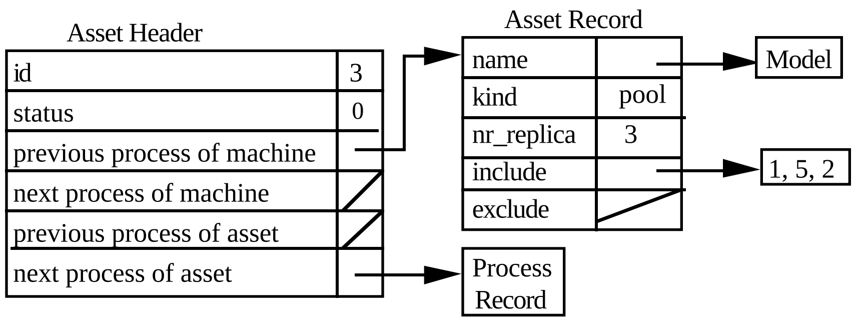


Figure 7.6. Asset Header Structure

A machine header also has two unused links compared to a normal process node. The field for process id is used to hold the index of the machine header, as in asset headers. The field for process status is used to hold the machine status. Machine status is different from process status. A machine may be in one of the three status: *allocated, deallocated* and *free*. The *link to the previous process of asset* field is coerced to a string pointer which references the name of the machine. The *link to the next process of asset* field is currently unused. It may be later used to point to a string describing the machine architecture type (for example, *sun3* or *sun4*, ... etc). The header structure is shown in Figure 7.7.

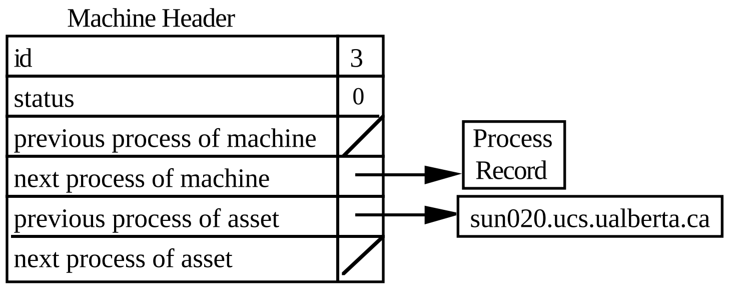


Figure 7.7. Machine Header Structure

7.6.2.2. Queues

In addition to the principle allocation map structure, the machine manager uses a number of queues to carry out the allocation/deallocation procedure. Each record in the queue contains an integer, which is the machine id. The structure is shown in Figure 7.8.

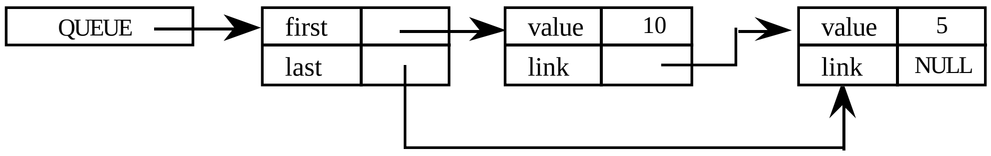


Figure 7.8. Queue Structure

Usually remove operations satisfy the first-in-first-out (FIFO) order as normal queues do, however, removal of arbitrary items from the queue is also supported. For example, if the *Enterprise* user has a specific choice of machines, the machines will not be removed in FIFO order from the free queue. The machine manager also uses the graph structure as discussed in Chapter 3. This is the same graph structure used in the textual interface.

7.6.2.3. Resource Secretary

The resource secretary is implemented as part of the machine manager as a temporary measure. The UNIX program *ruptime* is used to obtain the load averages on all the interested machines. The program ruptime can output if the machine is currently up or down, and three integers that indicate the load averages on the machines during the last 1, 5 and 15 minutes. If the load average over the last 1 minute is below a certain threshold, it is marked as usable. This mechanism has the disadvantage of using a magic threshold to determine if the machine is heavily loaded. The result is piped (on UNIX pipes) to the machine manager.

7.6.2.4. Launching Remote Processes

A simplistic reverse depth first traversal is used to launch processes on the allocated machines. The launching is started from the leaves of the call tree, so that a module used by other module will be put on the system before the modules using it. This will enable the acquaintance amongst the modules to be established as soon as possible. When launching processes on remote machines, the UNIX routine `system` was used initially. However, this routine forks off a UNIX shell which then executes the required command. Running the UNIX shell is an unnecessary step and it costs both time and memory. It doubles the number of processes created for each command executed, which is an unacceptable overhead. The memory and time required to run the UNIX shell depends on the shell program the user is running, and the complexity of his / her shell start up script, sometimes the cost can be substantial. In addition to run the shell, it also costs to duplicate the address space of the parent. As mentioned in Section 7.2 of this chapter, ISIS processes are very large. It is always not feasible to clone an ISIS process. It can happen that the operating system simply runs out of process entry space or memory to run the number of *Enterprise* processes required. Therefore, instead of using the `system` command, the machine manager simply `vfork's` a child and calls `exec` to launch the processes directly. The command `vfork` is used because the address space is not copied when spawning the child process. The issue of cost trade-offs in using the system call `system`, `fork`, and `vfork` are usually not a concern for most system programmers. This work presents a special case in which these implementation choices play an important role.

7.6.3. Implementation of Execution Manager

The execution manager interfaces the kernel to the other components in the overall *Enterprise* architecture, such as the code librarian and application manager. It accepts an *Enterprise* asset graph which is the data that communicates between the executive and the application manager. Although the *Enterprise* graph is actually used in the machine manager, the data structure to implement the graph is discussed in this section, since the Execution Manager serves as the front end to accept the graph.

```
department 3 A B C
line 2 B D
contract C
pool 3 D
service E
service F
A
   library  table.o queue.o -lm
B
   include  sass-lake, ardmore, bellis
   exclude  sundog
```

Figure 7.9. A Sample *Enterprise* Graph

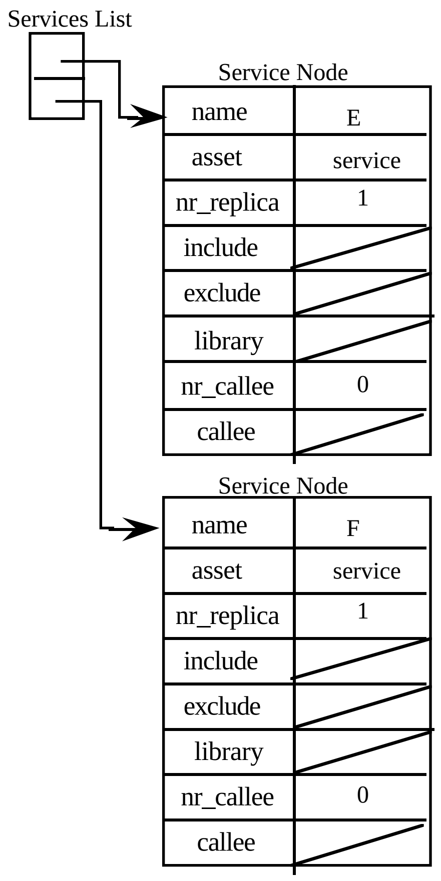


Figure 7.10. Service Structure of the *Enterprise* Application in Figure 7.8.

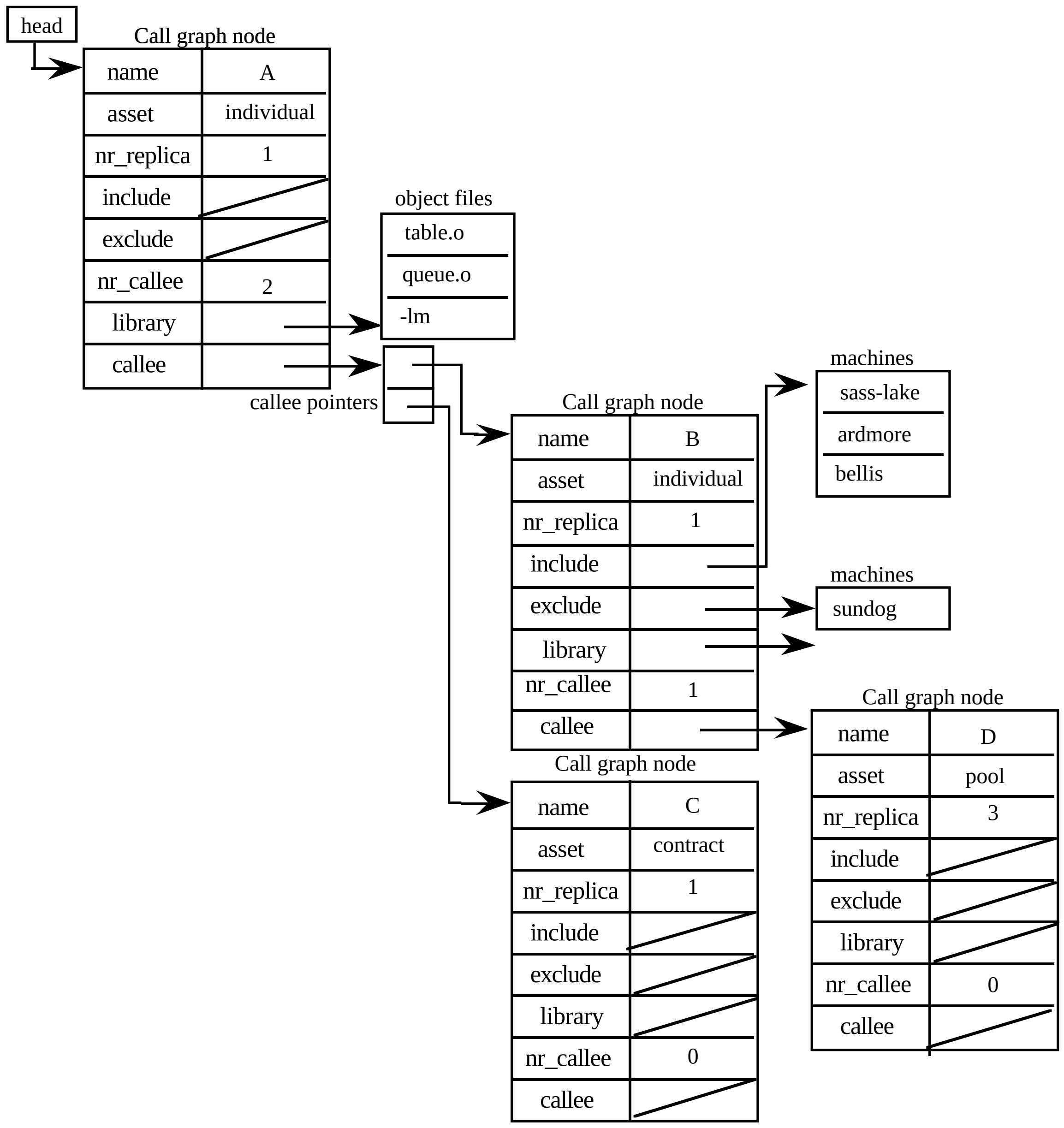


Figure 7.11. Call Graph Structure of the *Enterprise* Application in Figure 7.8

The program in Figure 7.9 serves only to illustrate its corresponding representation using the internal graph structure. A node in the call graph is identical to a node in the service list, although the overall organization is different. This example also demonstrates the implication of various coercions in *Enterprise* in terms of the call structure they generate.

7.7. Application Independent Experiments

These experiments were done to investigate the cost of executing *Enterprise* calls with varying message sizes. They offered insight into the performance characteristics and overheads of *Enterprise*. These results are not intended to be comprehensive but to investigate the kind of applications that can achieve speedup by using *Enterprise*. A sample of these applications, described in Appendix A, were tested to confirm the findings in this section.

The application independent results would also provide some suggestions for the relative performance characteristics of programming using various *Enterprise* assets. Although conceptually there are more than two kinds of assets, physically, *Enterprise* processes have only two configurations: single and replicated. Assets like pools, contracts and divisions are supported at run time by a group of replicated worker processes. Module calls made to a group of replicated workers are performed via a forwarder to ensure the job is taken by an idle worker. Module calls to a single process are made directly. By investigating the cost of these two kinds of calls, the cost of different kinds of assets are demonstrated. The experiments used to investigate *FrameWorks* were repeated here so that a comparison could be made. Since all *Enterprise* calls are asynchronous, only the p-call was experimented. The cost of an f-call would differ only by the extra cost of the reply. The time taken for 100 calls was measured for the two call configurations. Each experiment was repeated five times and the average of the observed times was calculated in each case. The results are shown in Figures 7.12.

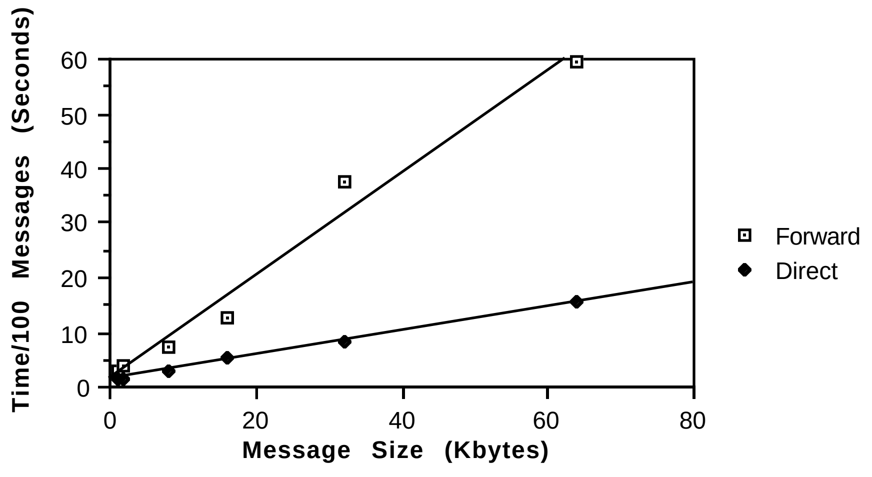


Figure 7.12 Cost of 100 *Enterprise* Calls

To compare *Enterprise* with the benchmark done by ISIS, the time required for a single call was also measured. Figure 7.13 shows the average time taken in executing a call as the message size is varied.

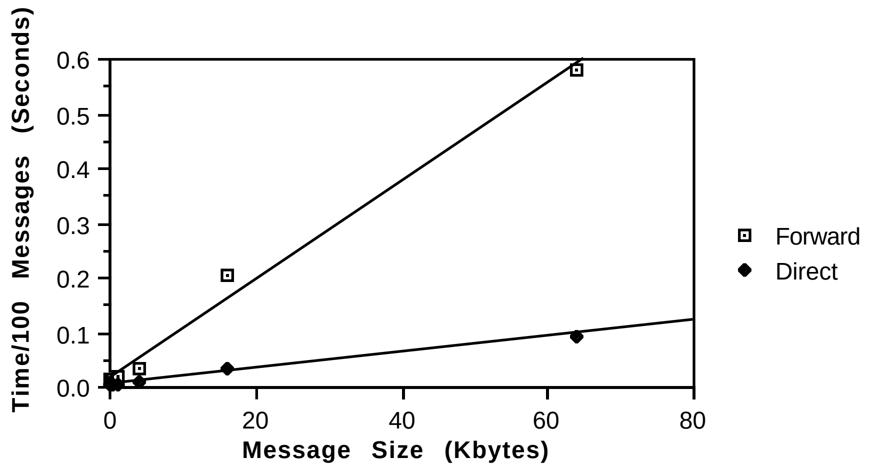


Figure 7.13 Cost of a Single *Enterprise* Call

Some useful observation can be made concerning the results. Figure 7.12 shows that the performance of doing 100 calls in *Enterprise* is similar to that in *FrameWorks.* A direct call in *Enterprise* is identical to a call in the pipeline configuration of *FrameWorks,* whereas a forwarded call in *Enterprise* is similar to a call in the manager configuration of *FrameWorks. Enterprise* is faster than *FrameWorks* in direct call and slower in call via a forwarder. These results are consistent with those reported by Birman, Schiper and Stephenson (1991). There has been no reported result on the performance of a call using the ISIS `forward` mechanism, however, intuitively the cost should be similar to that required in multicasting a call to two destinations, based on the amount of message exchange required. At the time of this writing, the `forward` mechanism in ISIS is not working completely and additional cost is incurred in an awkward workaround. In *Enterprise,* parameters in a call are sent directly as ISIS messages. There is no control information piggybacked on the message and so no additional cost is incurred. However, it should be noted that ISIS puts 220 bytes of header information on each packet transmitted. The same overhead would be required by the user if the program is written directly using ISIS code to achieve similar functionality as that generated by *Enterprise.*

The results in Figures 7.12 and 7.13 also suggest that calling via a forwarder is expensive. They represent the cost of a call forwarded to a single destination. When forwarding to multiple destinations, as in a pool or contract, the cost multiplies. It is a rule in parallel processing that adding more processor will not result in additional speedup unless each processor can work on a task of sufficient granularity. Figure 7.11 can be used as a guideline for the extent to which a program should be parallelized. Because of the high cost of communication in calls using a forwarder, in the *Animation* program given in Appendix A.1, coercing the *Spilt* asset to a pool of three process did not yield significant speedup. On the other hand, large grain computational intensive applications with grain size in the order of several seconds and even minutes are not uncommon in practice. For instance, the *Parallel* π and *Parallel Prime Number* computation in Appendices A.2 and A.3 have the grain sizes that are well within the range of good performance for the pool assets.

Birman, Schiper and Stephenson (1991) also give a detailed analysis of ISIS' performance and a comparison with SUN RPC. It has been reported that a null 2-process remote procedure call in ISIS had a round-trip latency of 6.51 ms. A 1 K message and a null reply took 7.86 ms. ISIS requires substantial CPU time in its creation of a lightweight task on the remote side, and recording the wall-clock time on every message sent or received. These represent significantly more work than those involved in other packages such as NMP or SUN RPC.

The results in Figures 7.12 and 7.13 also show that cost of communication grows quickly with the message size. ISIS is implemented using UDP, instead of TCP, because of TCP's small limit on the number of open connection at any time. However, the UNIX UDP implementation fails to handle high volume traffic. For example, having a hundred 4000-8000 bytes messages in transit at one time will actually cause UNIX to lose both outgoing and incoming UDP packets silently. ISIS uses a heuristic flow control which chokes the sender just before it is expected to overload UNIX. On the other hand, TCP is no better in handling high volume communication, as it is also limited to a buffer size of 52 Kbytes (Stevens, 1990). Therefore, applications that generate voluminous traffic will not benefit from using *Enterprise,* which is implemented on ISIS. An application of this kind not only runs up a huge "bill" in terms of CPU cycles needed to generate the message, but also overloads UDP and chokes the system if the destination does not consume the data out at a reasonable rate.

7.8. Conclusions

This chapter described the implementation of the *Enterprise* executive. It presented a justification and analysis of the implementation. Several implementation trade-offs were considered, for example, ISIS tasking verses direct message receive. These experiences should provide useful references to other developers who use ISIS as their implementation tool. The practical feasibility of the implementation was evaluated using a set of experiments. For large grain applications, *Enterprise* should be a platform for achieving good speedup. In contrast to a automatic parallelization tool, an *Enterprise* user still has full control over the parallel structure used in the application through the application graph.

Chapter 8

Conclusions and Future Research

8.1. Conclusions

This thesis demonstrates that the *Enterprise* system is worthwhile for developing distributed application. The thesis research consists mainly of two parts, the model and the realization of the model. The first half of the thesis describes the model and how it can be used to develop applications. The product of the research is a textual interface. The second half of the thesis presents the approaches to achieve the model. It contains a discussion of the design and implementation of the executive. The thesis concludes with the development of an application using *Enterprise* and a discussion of the performance achieved.

The first half of the thesis focuses on the programming model. *Enterprise* is distinct from other distributed programming environment in many aspects. *Enterprise* supports the unique programming philosophy of parallelization by hierarchical coercions. In many parallel or distributed computing environment, the user needs to draw communication graphs which usually involves drawing a diagram to connect nodes (processes) with arcs (communication paths). In *Enterprise,* the user is spared from these chores, instead, coercing and expanding nodes are all that required. Chapter 1 presented this design philosophy. Each kind of coercion represents a high level parallelization technique and is represented by an entity known as *asset*. *Enterprise* uses the analogy of an organization to illustrate each technique supported, for example, line and department. This analogy model is presented in Chapter 3, which describes the semantics of each coercion method in terms of data/control flow. In general, parallelism can be achieved by replication and partition of computation process, which are specified in *Enterprise* by hierarchical coercions. A textual user interface has been developed to support this programming concept. The feasibility of the programming model has been experimented by constructing several program using the environment, including an Animation program .

The other half of the thesis presents the design and implementation of the *Enterprise* executive. The system is developed using the ISIS distributed library package. An object-oriented methodology, which supports hierarchical design and program reuse, is used in the design. The experience from *FrameWorks* (Singh, 1991) proved useful in many areas of the development of *Enterprise.* The execution model was presented in Chapter 6 and 7. This model has the major advantage that coercing an asset will change only the asset itself since its internal composition is transparent to other assets. The model also ensures the proper handling of many concurrency issues such as synchronization, fairness and termination. For example, an *Enterprise* user will be relieved from the tedium of terminating any run-away process. Throughout the design and implementation, the issue of performance has always been a major concern, since the environment is aimed at supporting the notion of "the network is the supercomputer". This includes the awareness of detailed implementation issues, like using the less usual style of ISIS programming without tasks. A set of experiments is performed to assess the usefulness of the system and it is concluded that *Enterprise* provides an easy way to achieve good speedup on coarse grain problems.

8.2. Enterprise - The Next Generation

The *Enterprise* system is at the stage of prototyping and many of the desired features are still missing in the system. The project is an ongoing research project and this thesis is just a snapshot on some of the components of the system. For instance, the graphical user interface and the debugger is not described in this thesis. This section suggests some of the directions that future research can explore.

1. Several components of the *Enterprise* system are implemented as temporary measures, including the textual interface. Currently the graph is described by text. The interface can be improved to use menus developed using the UNIX *curses* screen handling package. Moreover, the syntax for specifying machine preferences can be extended to use query type descriptions such as 'memory > 8M'.

2. Another component of the *Enterprise* system which has been implemented as an interim measure is the resource secretary. In this thesis, the secretary is simply the UNIX *ruptime* program. Research is currently underway to construct a more sophisticated resource secretary (Theimer and Lantz, 1989).

3. As discussed in Chapter 7, implementing the communication primitives using ISIS libraries imposes several limitations. The most important limitation may be that users may need to acquire ISIS before they can use *Enterprise.* In general, any communication library can be used. Currently, the major ISIS features that *Enterprise* utilizes, in their order of extent, are process grouping, transparent global naming, message addressing by selectors, and concurrent tasks. Similar features are also supported by other distributed systems such as PVM (Sunderam, 1990) which can be used as an alternative communication backend.

4. Currently a simplistic depth-first traversal of the call tree is used to bind processes to machines. No effort has been made to map a process' resource utilization characteristics with the processor's capability. Processor allocation has always been an area of intensive research. A suitable heuristic would be to collect run time statistics of the application and use this information to improve the process allocation in the next run (Yan and Lundstrom, 1989).

5. The system described in this thesis executes *Enterprise* applications on a homogeneous network of machines. However, research is taking place (Chan, 1992) to investigate the extension of the current system to support a heterogeneous environment using the *Imake* program (DuBois, 1990).

6. At this moment *Enterprise* does not have sophisticated support for source code control or compilation on demand. The system will recompile all the assets in an *Enterprise* graph unless the user specifies a particular asset. Currently the *Enterprise* graph can be constructed using any text editor, it is not easy to determine from the graph if the user has coerced an asset. Coercion does not require the modification of the associated code, unless the call structure is altered. Therefore it is not possible to decide which asset is coerced by looking at the modification date of its source code file. The graphical editor, however, can support this feature since any modification to the graph goes through the *Enterprise* graphical interface.

7. Several new asset kinds are under construction. Many parallel/distributed programming environments support a *pardo* (or parallel for-next) construct in which the results are retrieved in the order which the RPC's are made. An asynchronous pool asset is being researched, in which items are sent off to compute asynchronously and the results can later be collected in a first-done-first-retrieved order. This would be similar to the concept of a bag in the sequential paradigm. In terms of execution, this asset is no different from an ordinary pool except that results should be polled instead of waited in a fixed order. The ISIS primitive `bc_pool` can be used to check if result from an RPC has been returned.

References

G.A. Agha. *ACTOR: A Model of Concurrent Computation in Distributed Systems*, 1986, MIT Press, Cambridge.

B. Appelbe and K. Smith. PAT: Interactive Conversion of Sequential to Parallel Fortran, *IEEE COMPCON*, 1990, pp. 585-588.

O. Babao$\overset{u}{g}$lu, L. Alvisi, A. Amoroso and R. Davoli. Paralex: An Environment for Parallel Programming in Distributed Systems, 1991, Technical Report UB-LCS-91-01, Department of Mathematics, University of Bologna, Bologna, Italy.

R.G. Babb. Parallel Processing with Large Grain Data Flow Techniques, *IEEE Computer*, 1984, vol. 17, no. 7, pp. 55-61.

R.G. Babb, II. *Programming Parallel Processors*, 1988, Addison Wesley, Reading.

H.E. Bal, J.G. Steiner and A.S. Tanenbaum. Programming Languages for Distributed Computing Systems, *ACM Computing Surveys*, 1989, vol. 2, no. 3, pp. 261-322.

G.M. Baudet.The Design and Analysis of Algorithms for Asynchronous Multiprocessors, 1978, Ph.D. Thesis, Computer Science Department, Carnegie-Mellon University.

A. Beguelin, J.J. Dongarra, G.A. Geist, R. Manchek and V.S. Sunderam. Graphical Development Tools for Network-Based Concurrent Supercomputing, in *Proceedings of Supercomputing 91*, 1991, pp. 435-444.

A. Beguelin, J.J. Dongarra, G.A. Geist, R. Manchek, K. Moore, R. Wade, J. Plank, and V. Sunderam. HeNCE: A User's Guide Version 1.2, 1992, Oak Ridge National Laboratory.

G. Bilardi. Some Observations on Models of Parallel Computation, in *Opportunities and Constraints of Parallel Computing*, 1989, Springer Verlag, New York, pp. 11-13.

K. Birman, T. Joseph, T. Raeuchle and A.E. Abbadi. Implementing Fault-Tolerant Distributed Objects, *IEEE Transactions on Software Engineering*, 1985, vol. SE-11, no. 6, pp. 502-508.

K. Birman, A. Schiper and P. Stephenson. Lightweight Causal and Atomic Group Multicast, *ACM Transactions on Computer Systems*, 1991, vol. 9, no. 3, pp. 272-314.

K. Birman. The Process-Group Approach to Reliable Distributed Computing, 1991, Technical Report TR-91-1216, Computer Science Department, Cornell University.

K. Birman, R. Cooper and B.Gleeson. Design Alternatives for Process Group Membership and Multicast, 1991, Technical Report TR-91-1257, Computer Science Department, Cornell University.

K. Birman. How to Securely Replicate Services, 1992, Technical Report TR-92-1274, Computer Science Department, Cornell University.

A. D. Birrell and B. J. Nelson. Implementing Remote Procedure Calls, *ACM Transactions on Computer Systems*, 1984, vol. 2, no. 1, pp. 39-59.

K.S. Booth, J. Schaeffer and W.M. Gentleman. Anthropomorphic Programming, 1984, Technical Report CS-82-47, Department of Computer Science, University of Waterloo.

G. Booch. Object-Oriented Development, *IEEE Transactions on Software Engineering*, 1986, vol. SE-12, no. 2, pp.211-221.

J.C. Browne. Formulation and Programming of Parallel Computations: A Unified Approach, in *Proceedings of the International Conference on Parallel Processing*, 1985, pp. 624-631.

J.C. Browne. Framework for Formulation and Analysis of Parallel Computation Structures, *Parallel Computing*, 1986, vol. 3, pp. 1-9.

J.C. Browne, M. Azam and S. Sobek. CODE: A Unified Approach to Parallel Programming, *IEEE Software*, 1989, vol. 6, no. 6, pp. 10-18.

J.C. Browne, T. Lee and J. Werth. Experimental Evaluation of a Reusability-Oriented Parallel Programming Environment, *IEEE Transactions on Software Engineering*, 1990, vol. 16, no. 2, pp. 111-120.

J.C. Browne, K. Sridharan, J, Kiall, C. Denton and W. Eventoff. Parallel Structuring of Real Time Simulation Programs, *IEEE COMPCON*, 1990, pp. 580-584.

J.C. Browne, J. Werth and T. Lee. Intersection of Parallel Structuring and Reuse of Software Components: A Calculus of Composition of Components for Parallel Programs, in *Proceedings of the International Conference on Parallel Processing*, 1989, pp. 126-130.

N. Budiraja, K. Marzullo, F.B. Schneider and S. Toueg. Primary-Backup Protocols: Lower Bounds and Optimal Implementations, 1992, Technical Report TR-92-1265, Computer Science Department, Cornell University.

N. Carriero and D. Gelernter. Applications Experience with Linda, in *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1988, ACM, New York.

N. Carriero and D. Gelernter. Linda in Context, *Communications of the ACM*, 1988, vol. 32, no. 4, pp. 444-458.

N. Carriero and D. Gelernter. How to Write Parallel Programs, *ACM Computing Surveys*, 1989, vol. 2, no. 3, pp. 323-357.

L. Chang and B.T. Smith. Classification and Evaluation of Parallel Programming Tools, 1990, Technical Report 1990-22, College of Engineering, University of New Mexico.

A. Chatterjee. Futures: A Mechanism for Concurrency Among Objects, in *Proceedings of Supercomputing '89*, 1989, pp. 562-567.

E. Chan, P. Lu, J. Mohsin, J. Schaeffer, C. Smith, D. Szafron and P.S. Wong. Enterprise: An Interactive Graphical Programming Environment for Distributed Software Development, 1991, Technical Report TR 91-17, Department of Computing Science, University of Alberta.

E. Chan. The Enterprise Librarian, 1992, M.Sc. Thesis, Department of Computing Science, University of Alberta, in preparation.

D. Cheriton and W. Zwaenepoel. Distributed Process Groups in the V Kernel. *ACM Transactions on Computer Systems*, 1985, vol. 3, no. 2, pp.77-107.

D. Comer. *Operating System Design: The XINU Approach*, 1984, Prentice-Hall, Toronto.

J.B. Dennis and E.C. Van Horn. Programming Semantics for Multiprogrammed Computations, *Communication of the ACM*, 1966, vol. 9, no. 3.

E.E. Dijkstra. *Selected Writings on Computing: A Personal Perspective,* 1982, Springer Verlag, New York.

D.C. DiNucci and R.G. Babb II. Architecture of the Parallel Programming Support Environment, *IEEE COMPCON*, 1989, pp. 102-107.

P. DuBois. Using Imake to Configure the X Window System Version 11 Release 4, 1990, Wisconsin Regional Primate Research Center.

M.R. Eskicioglu. Design Issues of Process Migration Facilities in Distributed Systems, *IEEE TCOS Newsletter*, 1990, vol. 4, no. 2, pp. 3-13.

S. I. Feldman. Make - A Program for Maintaining Computer Programs, *Software - Practice and Experience*, 1979, vol. 9, pp. 255-265.

W. Fenton, B. Ramkumar, V.A. Saletore, A.B. Sinha and L.V. Kalé. Supporting Machine Independent Programming on Diverse Parallel Architectures, in *Proceedings of the International Conference on Parallel Processing*, 1991, pp. 193-201.

M. J. Flynn. Very high-speed computing systems. *in Proceedings of the IEEE*, 1966, vol. 54, pp. 1901-1909.

J. Foley, A. van Dam, S. Feiner and J. Hughes. Computer Graphics: Principles and Practice, 2nd edition, 1990, Addison Wesley, Reading.

G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon and D. Walker. *Solving Problems on Concurrent Processors*, 1988, Prentice Hall, Englewood Cliffs.

G.A. Geist and V.S. Sunderam. Experiences with Network Based Concurrent Computing on the PVM System, 1991, Report Number TM-11760, Oak Ridge National Laboratory.

J. Gettys, P.L. Karlton and S. McGregor. The X Window System, Version II, *Software - Practice and Experience*, 1990, vol. 20, no. s2, pp. s35-s67.

W. Harrison. Tools for Multiple-CPU Environments, *IEEE Software*, 1990, vol. 7, May, pp. 45-51.

P.J. Hatcher, M.J. Quinn, A.J. Lapadula, R.K. Seevers, R.J. Anderson and R.R. Jones. Data-Parallel Programming on MIMD Computers, *IEEE Transactions on Parallel and Distributed Systems*, 1991, vol. 3, no. 2, pp. 377-383.

W.D. Hillis and G.L. Steele, Jr. Data Parallel Algorithms, *Communication of the ACM*, 1986, vol. 29, no. 12, pp. 1170-1180.

K. Hwang and F. Briggs. *Computer Architecture and Parallel Processing,* 1984, McGraw Hill, New York.

ISIS. The ISIS Distributed Toolkit, Version 3.0 User Reference Manual, 1992, ISIS Distributed Systems, Ithaca.

R. Jagannathan, A.R. Downing, W.T. Zaumen and R.K.S. Lee. Dataflow-based Methodology for Coarse-Grain Multiprocessing on a Network of Workstations, in *Proceedings of the International Conference on Parallel Processing,* 1989, pp. 54-58.

C.T. King, W.H. Chou and L.M. Ni. Pipelined Data-Parallel Algorithms: Part I-Concept and Modeling, *IEEE Transactions on Parallel and Distributed Systems,* 1990, vol. 1, no. 4, pp. 470-485.

H.T. Kung. Lets Design Algorithms for VLSI Systems, in *Proceedings of the Caltech Conference on Very Large Scale Integration*, 1979, pp. 65-90.

T.G. Lewis and W.G. Rudd. Architecture of the Parallel Programming Support Environment, *IEEE COMPCON,* 1990, pp. 589-594.

X. Li, P. Lu, J. Schaeffer, J. Shillington, P.S. Wong and H. Shi. On the Versatility of Parallel Sorting by Regular Sampling, 1992, Technical Report TR 91-06, Department of Computing Science, University of Alberta.

T.A. Marsland, T. Breitkreutz and S. Sutphen. A Network Multi-processor for Experiments in Parallelism, *Concurrency: Practice and Experience*, 1991, vol. 3, no. 3, pp. 203-219.

Myrias. Parallel Programmer's Guide, Myrias Research Corporation, 1990, Edmonton.

D.A. Nichols. Using Idle Workstations in a Shared Computing Environment, in *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, 1987, pp. 5-12.

C.M. Pancake and S. Utter. Models for Visualization in Parallel Debuggers, in *Supercomputing '89,* 1989, pp. 627-636.

J.L. Peterson and A. Silberschatz. *Operating System Concepts,* Alternate Edition, 1985, Addison Wesley, Reading.

J. Schaeffer. Distributed Game-Tree Searching, *Journal of Parallel and Distributed Computing,* 1989, vol. 6, pp. 90-114.

R.W. Scheifler and J. Gettys. The X Window System, *ACM Transactions on Graphics*, 1986, vol. 5, pp. 79-109.

Z. Segall and L. Rudolph. Pie (A Programming and Instrumentation Environment for Parallel Processing), *IEEE Software*, 1985, vol. 2, no. 6, pp. 22-37.

A. Singh. A Template-Based Approach to Structuring Distributed Algorithms Using a Network of Workstations, 1991, Ph.D. Thesis, Department of Computing Science, University of Alberta.

A. Singh, J. Schaeffer and M. Green. Structuring Distributed Algorithms in a Workstation Environment: The FrameWorks Approach, in *Proceedings of the International Conference on Parallel Processing,* 1989, pp. 89-97.

A. Singh, J. Schaeffer and M. Green. A Template-Based Tool for Building Applications in a Multicomputer Network Environment, in *Parallel Computing*, 1989, D. Evans, G. Joubert and F. Peters (editors), North-Holland, pp. 461-466.

A. Singh, J. Schaeffer and M. Green. A Template-Based Approach to the Generation of Distributed Applications Using a Network of Workstations, *IEEE Transactions on Parallel and Distributed Systems,* 1991, vol. 2, no. 1, pp. 52-67.

J.P. Singh, W-D. Weber and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory, 1991, Technical Report CSL-TR-91, Computer Systems Laboratory, Stanford University.

K. Smith and B. Appelbe. PAT-An Interactive Fortran Parallelizing Assistant Tool, in *Proceedings of the International Conference on Parallel Processing*, 1988, pp. 58-62.

K. Smith, B. Appelbe and K. Stirewalt. Incremental Dependence Analysis for Interactive Parallelization, in *Supercomputing '90* , 1990, pp. 330-341.

S. Sobek, M. Azam and J.C. Browne. Architecture and Language Independent Parallel Programming: A Feasibility Demonstration, in *Proceedings of the International Conference on Parallel Processing*, 1988, pp. 80-83.

K. Sridharan, M. McShea, C. Denton, B. Eventoff, J. Browne, P. Newton, M. Ellis, D. Grossbard, T. Wise and D. Clemmer. An Environment for Parallel Structuring of Fortran Programs, in *Proceedings of the International Conference on Parallel Processing*, 1989, pp. 98-106.

K. Sridharan, R. Narayanaswamy, C. Denton and B. Eventoff. Parallel Structuring of Programs Containing I/O Statements, in *Proceedings of the International Conference on Parallel Processing*, 1990, pp. 224-228.

W.R. Stevens. *UNIX Network Programming*, 1990, Prentice Hall, Englewood Cliffs.

B. Stroustrup. What is "Object Oriented Programming", *IEEE Software,* 1988, pp. 10-20.

B. Sugla, J. Edmark and B. Robinson. An Introduction to the CAPER Application Programming Environment, in *Proceedings of the International Conference on Parallel Processing*, 1989, pp. 107-111.

P.A. Suhler. Heuristic Tuning of Parallel Loop Performance, in *Proceedings of the International Conference on Parallel Processing*, 1989, pp. 184-191.

P.A. Suhler, J. Biswas and K.M. Korner. TDFL - A Task-Level Data Flow Language, 1987, Technical Report TR-87-44, Department of Computer Science, University of Texas at Austin.

P.A. Suhler, J. Biswas, K.M. Korner and J.C. Browne. TDFL: A Task-Level Dataflow Language, *Journal of Parallel and Distributed Computing*, 1990, vol. 9, no. 2, pp. 103-115.

D. Szafron, J. Schaeffer, P.S. Wong, E. Chan, P. Lu and C. Smith. Enterprise: An Interactive Graphical Programming Environment for Distributed Software, in *Proceedings of the Conference on Programming Environments for Parallel Computing*, 1992, to appear.

V.S. Sunderam. PVM: A Framework for Distributed Computing, *Concurrency: Practice and Experience*, 1990, vol. 2, no. 4, pp. 315-339.

M.M. Theimer and K.A. Lantz. Finding Idle Machines in a Workstation-Based Distributed System, *IEEE Transactions on Software Engineering*, 1989, vol. 15, no. 11, pp. 1444-1453.

D. Vrsalovic, Z. Segall, D. Siewlorek, F. Gregorettl, E. Caplan, C. Fineman, S. Kravltz, T. Lehr and M. Russinovich. Performance Efficient Parallel Programming in MPC, 1988, Technical Report CMU-CS-88-164, Department of Computer Science, Carnegie Mellon University.

Z. Xu and K. Hwang. Molecule: A Language Construct for Layered Development of Parallel Programs, *IEEE Transactions on Software Engineering*, 1989, vol. 15, no. 5, pp. 587-599.

J.C. Yan and S.F. Lundstrom. The Post-Game Analysis Framework - Developing Resource Management Strategies for Concurrent Systems, *IEEE Transactions on Knowledge and Data Engineering*, 1989, vol. 1, no. 3, pp. 293-309.

D.A. Young. *OSF/Motif Reference Guide,* 1990, Prentice Hall, Englewood Cliffs.

D.A. Young. *The X Window System Programming and Application with Xt OSF/Motif*, 1990, Prentice Hall, Englewood Cliffs.

Appendix A

Some Example Applications

This appendix presents a few simple applications constructed and executed using the *Enterprise* system. The hardware computing environment has a number of Sun4 workstations connected over a *Ethernet*-based local area network. The examples does not cover all the features and capabilities of *Enterprise*, but demonstrate the basic idea of the system.

A.1. The Animation Application

More experimental results are performed using the *Animation* Application described in Chapter 4. In addition to the sequential version, two distributed configurations discussed in Chapter 3 were implemented. These configuration are shown in Figures A.1 and A.2.

```
line 3 Model PolyConv Split
Model
        library -lUTILITY -lfb -lm
PolyConv
        library -lUTILITY -lfb -lm
Split
        library -lUTILITY -lfb -lm
```

Figure A.1. *Animation* as a Line of Three

```
line 3 Model PolyConv Split
pool 5 Split
Model
        library -lUTILITY -lfb -lm
PolyConv
        library -lUTILITY -lfb -lm
Split
        library -lUTILITY -lfb -lm
```

Figure A.2. *Animation* as a Line of Three and *Split* as Pool

For this program, a speedup of 1.75 was obtained by using a line running on three processors instead of a single individual asset (a sequential program). This result is slightly better than that achieved by *FrameWorks*. Note that any timings are subject to large variations, depending on the number of available processors and the amount of traffic on the network. In this particular case, the timings were done late at night with minimal contention with other users on processors and network traffic.

By coercing the *Split* asset to a pool of 3 processes and running each process on a different processor, a speedup of 2.1 was obtained. It is important to keep in mind that no change to the source code is necessary, and only the *Split* module needs to be recompiled.

The example shows that experimentation with different configuration of processors can be easily performed in *Enterprise*. The application is able to achieve a fairly good speedup, yet the absolute value is not important. Since the application would otherwise be executed sequentially, *any* improvement in performance would be welcomed. The fact that this speedup can be achieved without any change to the sequential code is the major advantage of using *Enterprise.* The parallel version involves much complex issues, for example, mutually exclusive selection of an idle processor, if to be constructed manually even using a high level distributed library package, such as ISIS or NMP. Writing it from scratch using sockets or RPC's would involve so much effort that makes it less feasible.

A.2. The Prime Number Generator

The application *Prime* implements a parallel program for finding all the prime numbers that falls within a given range. The application consists of two modules: *Partition* and *FindPrimes*. *Partition* obtains from input parameters the interval to be searched and the number of subranges to divide the search domain. It computes the subranges to be search and sets up the output format for the results. This information is then passed to the module *FindPrimes* via a function call.

The parallel program is a line asset of two members: *Partition* and *FindPrimes*, as shown in Figure A.3. *FindPrimes* is a pool asset which has 2 members. Each member runs the code of the *FindPrimes* module and all replicas run in parallel on 2 workstations. The prime numbers generated are written to a file.

```
line 2 Partition FindPrimes
pool 2 FindPrimes
Partition:
   library -lm
FindPrimes
   library -lm
```

Figure A.3. Application Graph for Prime

Prime numbers in the range 0 to 1,000,000 were searched. The range was divided into 20 subranges. The sequential application took 426 seconds to execute on a Sun4. This is an application that is relatively easy to achieve good speedup because of its high granularity and low communication needs. The experiment was repeated with pool sizes of 4 and 8, which required no change or recompilation of code at all. The speedup curve is shown in Figure A.4.

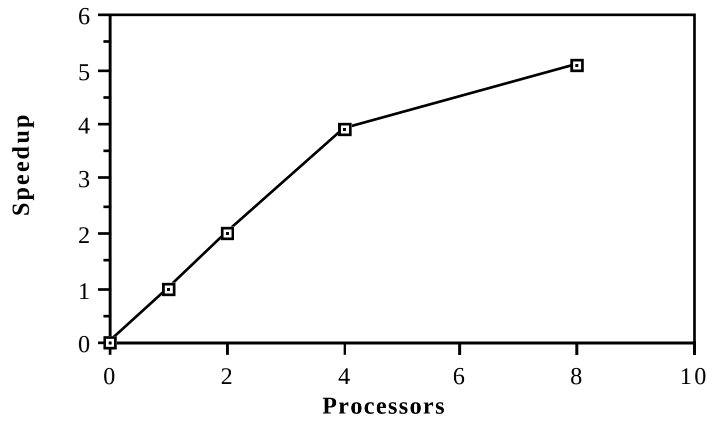


Figure A.4. Speedup for Prime

The knee in the speedup curve shows that when a pool of 8 workers is used to run *FindPrimes*, the gain in parallelism can no longer as effectively offset the communication overhead incurred as with other configurations. The experiment was also repeated by using a contract to run the *FindPrimes* module, and a speedup of 3.6 was attained.

```
line 2 Partition FindPrimes
contract FindPrimes
Partition:
   library -lm
FindPrimes
   library -lm
```

Figure A.5. Application Graph for Prime using dynamic processes

A.3. Parallel π Computation

We illustrate a practical application of *Enterprise* using a common parallel numerical analysis application. The *Parallel π* program has been used to demonstrate many parallel programming environments (Babb, 1988; Hatcher et al., 1991). The application computes the value of π by numerically calculating the integral:

$$\int_0^1 \frac{4}{(1+x^2)} dx = \tan^{-1}(1) = \pi$$

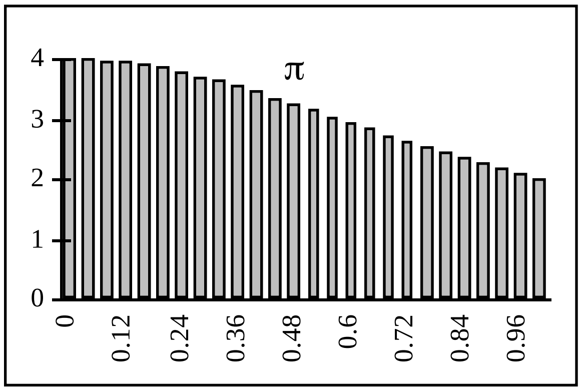


Figure A.5. The area under the curve $4/(1+x^2)$ gives π

This is the area under the curve of Figure A.4. The areas under the vertical strips are calculated in parallel. Area of each stripe is approximated by estimating the areas under each of thousands of very narrow stripes. The program is usually done using a master-slave construct in which each slave computes a segment of the area. Each slave returns its portion of the area and is accumulated to obtain the final value of π.

The application graph for *Parallel π* is shown in Figure A.5. The program is a line asset of two members: *Partition* and *FindPi*. *FindPi* is a pool asset which has 3 members. The application is similar in its parallel graph structure to the *Prime* application but function calls, instead of procedure calls, are used in the modules. This should illustrate how calls that expect replies perform.

```
line 2 Partition FindPi
pool 3 FindPi
Partition:
   library -lm
FindPi
   library -lm
```

Figure A.6. Application Graph for Pi

In one experiment, the area was divided into 90,000 stripes. Each member ran the code of the *FindPi* module and all replicas ran in parallel on 3 workstations. The sequential application took 1.057 seconds to execute on a Sun4. The parallel computation achieved a 1.74 fold speedup.

A.4. Chaotic Gauss-Seidel

We present an implementation of Baudet's (1978) parallelization of the Gauss Seidel algorithm for solving the family of equations $Ax = b$. The program is divided up into three sections. The first section receives input from the user and then divides up the work and distributes it to the processors. The second section serves as shared memory and responds to requests from other routines to either initialize x, update a portion of the x vector, or return the current value of the x vector. This section is implemented as a service. The third section asks the service for the current state of the x vector and iterates its section of the vector until the tolerance is met. A pool is used to run this section and all segments run in parallel. The application graph for this application is shown in Figure A.7.

```
line 2 baudet doTheWork
pool 10 doTheWork
service serviceXValues
baudet
        library -lm
doTheWork
        library -lm
serviceXValues
        library -lm
```

Figure A.7. Application Graph for Chaotic Gauss-Seidel

In one experiment with a problem size of 1000, a speedup of at least half-linear was achieved, as Figure A.8 shows.

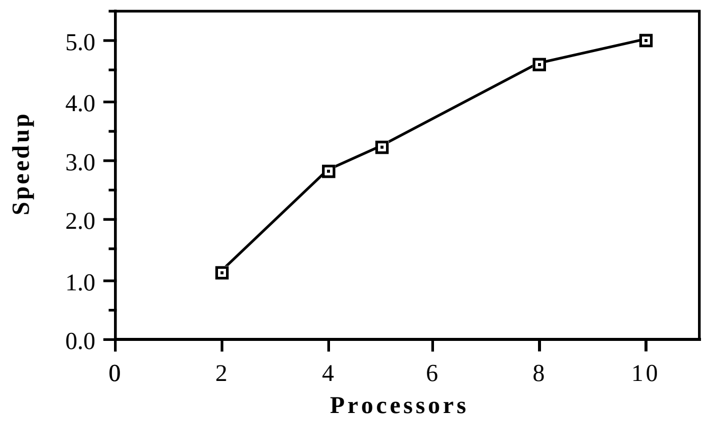


Figure A.8. Speedup for Chaotic Gauss-Seidel

The above examples demonstrates several important advantages of *Enterprise*:

1. These examples show that worthwhile performance gains are possible using the *Enterprise* system.

2. All the applications can be modified to run from existing sequential program with little modification (splitting of code into different files). The code modification performed by *Enterprise* on the *Animation* application can be found in Appendix C, with the original code shown in Appendix B.

3. Different configuration of each of the application can be generated with no change of code at all.

Appendix B.

C Code for the Entry Procedures of the Animation Example

This appendix gives pseudo code for the *Animation* example. For brevity, only the main procedure calls of *Model*, *PolyConv* and *Split* are shown.

Asset Code: *Model*

```
/* Model asset */

#define    NUMBER_STEPS    4
#define    NUMBER_FISH       10
#define    NUMBER_FRAMES  20

main()
{
    float timeperframe;
    int frame;

    /* Generate the school of fish */
    MakeFish( NUMBER_FISH, 0 );

    /* Loop through each frame */
    timeperframe = 1.0 / NUMBER_STEPS;
    for( frame = 0; frame < NUMBER_FRAMES; frame++ )
    {
        /* Do model computations */
        InitModel( NUMBER_FISH );
        MoveFish( NUMBER_FISH, timeperframe );
        DrawFish( NUMBER_FISH, timeperframe * frame );
        WriteModel( frame );

        /* Done!  Send work to PolyConv process */
        PolyConv( frame );
    }
}
```

Asset Code: *PolyConv*

```
/* PolyConv asset */

#define MAX_POLYGONS 1000

PolyConv( frame )
int frame;
{
   polygon polygontable[ MAX_POLYGONS ];
   int npoly;

   /* Convert polygons and send to Split */
   DoConversion( frame );
   npoly = ComputePolygons( &polygontable );
   Split(frame, npoly, polygontable);
}
```

Asset Code: *Split*

```
/* Split asset */

#define    MAX_POLYGONS   1000

Split(frame, npoly, polygontable)
int frame, npoly;
polygon polygontable;
{
   HiddenSurface( frame, npoly, polygontable );
   AntiAlias( frame, npoly, polygontable );
}
```

Appendix C.

Entry Procedures of the Animation Example with Enterprise Code

The Animation application in Chapter 2, along with the *Enterprise* diagram in Figure A.2, is translated automatically into an executable program. In this Appendix, the program that *Enterprise* produces is given. In the code, *italic* text refers to inserted code, while regular text identifies the user-written code. To limit the illustration within a reasonable space, this example serves to demonstrate what *Enterprise* does to user code, but not the complete accurate code being inserted.

This illustration also shows that most of the *Enterprise* code is inserted as a block of initialization code, including the code to establish communication. The *Enterprise* code for *Split*, which is a pool and *PolyConv*, which is an individual, is similar except in a few statements in their initialization code. In other words, coercing an asset to a different kind requires simply linking the module with a different initialization code template.

Asset Code: *Model*

```
#include "isis.h"
#include <signal.h>

/* Declare services entry numbers */
#define _e_CALL   11

/* Declare constants and global flags */
#define _e_REPLY  1
#define _e_NREPLY 0
int _e_busy = FALSE;

/* Declare address of all called modules */
address *_e_PolyConv;

/* Establish connection with ISIS and its callee(s) */
_e_init()
{
  isis_remote_init(0,0,0,ISIS_NOCOPYRIGHTNOTICE);
  do
    _e_PolyConv = pg_lookup("PolyConv");
  while (addr_isnull(_e_PolyConv));
  pg_client(_e_PolyConv,"PolyConv");
  isis_start_done();
}

#define    NUMBER_STEPS   4
#define    NUMBER_FISH       10
#define    NUMBER_FRAMES  20

main( )
{
    float timeperframe;
    int frame;

    /*** Enterprise code inserted for main() ***/
    _e_init ();

    /* Generate the school of fish */
    MakeFish( NUMBER_FISH, 0 );
    /* Loop through each frame */
    timeperframe = 1.0 / NUMBER_STEPS;
    for( frame = 0; frame < NUMBER_FRAMES; frame++ )
    {
        /* Do model computations */
        InitModel( NUMBER_FISH );
        MoveFish( NUMBER_FISH, timeperframe );
        DrawFish( NUMBER_FISH, timeperframe*frame );
        WriteModel(frame);

        /* Done!  Send work to PolyConv process */
        /*
         *  Function call of PolyConv()
         *              PolyConv(frame);
         */
        bcast(_e_PolyConv,_e_CALL,"%d",frame,_e_NREPLY);
    }
}
```

Asset Code: *PolyConv*

```
#include "isis.h"

/* Declare services entry numbers */
#define _e_FINISH 1
#define _e_ABORT  2
#define _e_CALL   11

/* Declare constants and global flags */
#define _e_REPLY  1
#define _e_NREPLY 0
int _e_busy = FALSE;

/* Declare address of itself and all called modules */
address *_e_my_pg;
address *_e_Split;
address *_e_manager;

/* This entry terminates the asset immediately */
_e_abort_handle(_e_msg_p)
message *_e_msg_p;
{
  msg_get(_e_msg_p,"");
  exit();
}

/* This entry terminates the asset when it has done all calls */
int _e_finish = FALSE;
message *_e_finish_msg;
_e_finish_handle(_e_msg_p)
message *_e_msg_p;
{
  msg_get(_e_msg_p,"");
  if (msg_ready(_e_CALL)==0 && !_e_busy) {
    reply(_e_msg_p,"%d",1);
    flush();
    exit();
  }
  else {
    _e_finish = TRUE;
    msg_increfcount(_e_msg_p);
    _e_finish_msg = _e_msg_p;
  }
}

_e_init()
{
  isis_remote_init(0,0,0,ISIS_NOCOPYRIGHTNOTICE);
  isis_entry(_e_CALL,MSG_ENQUEUE,"enqueue");
  isis_entry(_e_FINISH, _e_finish_handle, "_e_finish_handle");
  isis_entry(_e_ABORT, _e_abort_handle, "_e_abort_handle");
  _e_my_pg = pg_join("PolyConv_worker",0);
  do
    _e_manager = pg_lookup("PolyConv");
  while (addr_isnull(_e_manager));
  do
    _e_Split = pg_lookup("Split");
  while (addr_isnull(_e_Split));
  pg_client(_e_Split,"Split");
  isis_start_done();
}

main()
{
  message *_e_msg_p;
  _e_init();
  while (1) {
    message *_e_wrapper_msg_p;
#define _e_CHECK_IN   12
    cbcast(_e_manager,_e_CHECK_IN,"%d",pg_rank(_e_my_pg,&my_address),_e_NREPLY);
    _e_wrapper_msg_p = msg_rcv(_e_CALL);
    msg_get(_e_wrapper_msg_p,"%m",&_e_msg_p);
    _e_busy = 1;
    PolyConv(_e_msg_p);
    _e_busy = 0;
    msg_delete(_e_msg_p);
    msg_delete(_e_wrapper_msg_p);
    if (_e_finish && (msg_ready(_e_CALL) == 0)) {
      reply(_e_finish_msg,"%d",1);
      msg_delete(_e_finish_msg);
      flush();
      exit(0);
    }
  }
}

/*
 * This is an Enterprise asset.  The original
 * declaration of the function is commented out
 * and is replaced by the inserted code.  The
 * code inserted depends on the asset's type.
 *
 * PolyConv(frame)
 */
PolyConv(msg_p)
message *msg_p;
{
int frame;
{
    int npoly;
    polygon polygontable;

    msg_get(msg_p,"%d",&frame);

    DoConversion(frame);
    /* Done!  Send work to PolyConv process */
    /*
     *  Function call of Split()
     *              Split(frame);
     */
    bcast(_e_Split,_e_CALL,"%d%d%C",frame,npoly,polygontable,sizeof(polygontable),_e_NREPLY);
    return;
}
}
```

Asset Code: *Split*

```
#include "isis.h"

/* Declare services entry numbers */
#define _e_FINISH 1
#define _e_ABORT  2
#define _e_CALL   11

/* Declare constants and global flags */
#define _e_REPLY  1
#define _e_NREPLY 0
int _e_busy = FALSE;

/* Declare address of itself and all called modules */
address *_e_my_pg;
address *_e_manager;

/* This entry terminates the asset immediately */
_e_abort_handle(_e_msg_p)
message *_e_msg_p;
{
  msg_get(_e_msg_p,"");
  exit();
}

/* This entry terminates the asset when it has done all calls */
int _e_finish = FALSE;
message *_e_finish_msg;
_e_finish_handle(_e_msg_p)
message *_e_msg_p;
{
  msg_get(_e_msg_p,"");
  if (msg_ready(_e_CALL)==0 && !_e_busy) {
    reply(_e_msg_p,"%d",1);
    flush();
    exit();
  }
  else {
    _e_finish = TRUE;
    msg_increfcount(_e_msg_p);
    _e_finish_msg = _e_msg_p;
  }
}

_e_init()
{
  isis_remote_init(0,0,0,ISIS_NOCOPYRIGHTNOTICE);
  isis_entry(_e_CALL,MSG_ENQUEUE,"enqueue");
  isis_entry(_e_FINISH, _e_finish_handle, "_e_finish_handle");
  isis_entry(_e_ABORT, _e_abort_handle, "_e_abort_handle");
  _e_my_pg = pg_join("Split_worker",0);
  do
    _e_manager = pg_lookup("Split");
  while (addr_isnull(_e_manager));
  isis_start_done();
}

main()
{
  message *_e_msg_p;
  _e_init();
  while (1) {
    message *_e_wrapper_msg_p;
#define _e_CHECK_IN   12
    bcast(_e_manager,_e_CHECK_IN,"%A[1]",&my_address,_e_NREPLY);
    _e_wrapper_msg_p = msg_rcv(_e_CALL);
    msg_get(_e_wrapper_msg_p,"%m",&_e_msg_p);
    _e_busy = 1;
    Split(_e_msg_p);
    _e_busy = 0;
    msg_delete(_e_msg_p);
    msg_delete(_e_wrapper_msg_p);
    if (_e_finish && (msg_ready(_e_CALL) == 0)) {
      reply(_e_finish_msg,"%d",1);
      msg_delete(_e_finish_msg);
      flush();
      exit(0);
    }
  }
}

#define    MAX_POLYGONS   10
/*
 * This is an Enterprise asset.  The original
 * declaration of the function is commented out
 * and is replaced by the inserted code.  The
 * code inserted depends on the asset's type.
 *
 * Split(frame,npoly,polygontable)
 */
Split(msg_p)
message *msg_p;
{
/* generating code for { */
{

    int npoly, frame;
    polygon polygontable;

    msg_get(msg_p,"%d%d%C",&frame,&npoly,&polygontable,NULL);

    HiddenSurface(frame,npoly,polygontable);
    AntiAlias(frame,npoly,polygontable);
}
}
```