Scheduling Support Mechanisms for Autonomous, Heterogeneous, Distributed Systems

A Thesis
Submitted to the Faculty

of

Purdue University

by

Stephen Joel Chapin

In Partial Fulfillment of the
Requirements for the Degree

of

Doctor of Philosophy

December 1993

For Laurie, who never had the chance.

ACKNOWLEDGMENTS

Many thanks to Douglas Comer, Michal Young, and Ahmed Elmagarmid for advice and for serving on my advisory committee; to Bharat Bhargava for suggesting applications of this work outside the domain of program scheduling; to my colleagues in the Shadow Lab for the good times and lengthy debates on minutæ; to Ryan Stansifer and Eugene Spafford for Quality advice; to the National Aeronautic and Space Administration for funding under Graduate Student Researchers Program grant number NGT 50919; to Herb, Joyce, Lee, and Betsy for years of love and support; to Nate for coming along at the right time and providing inspiration; and most especially, to Mary, for being all that she is.

"I know how the flowers felt."

—Robert Frost, "Lodged"

TRADEMARKS

NetShare is a trademark of Aggregate Computing, Inc.

UNIX is a trademark of Unix Systems Laboratories.

VMS is a trademark of Digital Equipment Corporation.

Load Balancer is a product of Freedman Sharp and Associates.

Sun 3, SPARC IPC, SPARC IPX, SPARCstation 1+, and SunOS are trademarks of Sun Microsystems, Inc.

MS-DOS is a trademark of Microsoft.

DISCARD THIS PAGE

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABSTRACT

Chapin, Stephen Joel. Ph.D., Purdue University, December 1993. Scheduling Support Mechanisms for Autonomous, Heterogeneous, Distributed Systems. Major Professor: Eugene H. Spafford

An essential component of effective use of distributed systems is proper task placement, or scheduling. To produce high-quality schedules, scheduling algorithms require underlying support mechanisms that provide information describing the distributed system. The work presented here makes a clear distinction between scheduling policies and the underlying mechanism, and focuses on the problem of providing general-purpose mechanisms that facilitate a broad spectrum of task placement algorithms.

This dissertation proposes a model for distributed scheduling support mechanisms. This model includes scalable and extensible mechanisms that support the efficient implementation of scheduling policies on distributed systems, while preserving the autonomy of the component systems. The mechanisms include provably correct information exchange protocols for system state dissemination in distributed systems.

MESSIAHS is a prototype implementation of these mechanisms, including a scheduling module that implements the basic mechanism, as well as a library of function calls and a specialized programming language for writing distributed schedulers. As a demonstration of the utility of the prototype, several algorithms from the literature are implemented and their performance is analyzed. The experimental results show average overhead of approximately 10% using MESSIAHS, measured against a theoretical ideal running time. The results indicate that it is possible to build scalable, general-purpose mechanisms that support a variety of task placement algorithms while preserving autonomy.

1. INTRODUCTION

A typical research environment represents a large investment in computing equipment, including dozens of workstations, several mainframe machines, and possibly a small number of supercomputers. Taken collectively, the aggregate computing resources are sufficient to solve difficult problems such as large-number factoring or climate simulation. When each machine is used in isolation, several limitations appear.

Gantz, Silverman, and Stuart [GSS89] and Litzkow [Lit87] show that equipment in a workstation-based environment is underutilized, with processor utilization as low as 30%. Although the combined resources of several machines might solve the problem at hand, users of this equipment often find that the resources local to each machine, such as memory, disk space, and processing power, are not sufficient to execute large programs (see Karp, Miura, and Simon [KMS93] for examples). Certain scientific application programs have distinct components, best suited for massively-parallel machines, vector-processing supercomputers, or graphics-visualization workstations. Restricting execution of all the components to one machine or executing components on inappropriate machines incurs delay that could be avoided if each component were executed on the architecture best suited for it.

A solution to these limitations is to conglomerate the separate processors into a *distributed system*. Distributed systems communicate by passing messages over an external communications channel. Such systems are often called loosely-coupled systems, in contrast to tightly-coupled parallel machines that communicate through shared memory [HB84]. Coupling represents only one quality of distributed systems. Enslow [Ens78] defines four aspects of distribution: hardware distribution, data distribution, processing distribution, and control distribution. Most distributed systems, especially those that assign programs to processors for execution, fail to exploit control distribution fully. Instead, these systems use a centralized control mechanism to manage distributed hardware, which often results in poor processing distribution.

Distributed systems can be joined into larger distributed systems to further expand the computational power of the whole. Software modules running on the individual computers can assign programs to processors for execution. However, obstacles such as incompatible architectures and restrictive administrative domains hinder the formation of large-scale distributed systems composed of autonomous, heterogeneous systems.

In a conventional situation, a user has to discover which processors are currently available, reserve them for computation, manually place the programs and associated data files on the machines, and serve as coordinator for their execution. Using an automated scheduling system, the user submits the individual programs to the scheduling system running on a host participating in the distributed system, the system automatically locates suitable execution sites and schedules the programs for execution.

This dissertation defines a *task* as a consumer of resources. Examples of tasks include the conventional model of a computationally intensive unit in a larger program, as well as a set of database queries (see Carey, et al. [CLL85]), output requests for printers, and data transfers over a communication network. For simplicity of description, this dissertation restricts further discussions to the conventional model of placing computational tasks on processors. A *task force* (as defined in [VW84]) comprises a group of cooperating tasks for solving a single problem.

Within a distributed system, there are two levels of task scheduling: the association of tasks with processors (global scheduling), and the choice of which task to execute among those available on a processor (local scheduling) [CK88]. This dissertation concentrates on developing support for global scheduling.

Webster's Dictionary defines *autonomous* as "having the power of self-government," or as "responding, reacting, or developing independently of the whole." Thus, an autonomous system makes local policy decisions and can act without the permission of any external authority. In autonomous systems, all information, behavior, and policy pertaining to a system are private to that system. Any disclosure of private information is at the discretion of the local system.

Because of the prevailing decentralization of computing resources, autonomy plays an increasingly important role in distributed computation systems. No longer does a single authoritative entity control the computers in a large organization. Users may control a few machines of their own, and their department may have administrative control over several such sets of machines. Their department may be part of a regional site, which is, in turn, part of a nationwide organization. No single entity, from the user to the large organization, has complete control over all the computers it may wish to use.

Garcia-Molina and Kogan [GMK88], and Eliassen and Veijalainen [EV87] have examined autonomy in distributed systems and devised taxonomies for different types of autonomy. The scheme proposed by Eliassen and Veijalainen is more general but less detailed than that proposed by Garcia-Molina. The following four classes of autonomy combine the two schemes and tailor the definitions to the application of distributed scheduling.

design autonomy

The designers of individual systems are not bound by other architectures, but can design their hardware and software to their own specifications and needs. *Heterogeneous systems* are multiprocessor systems that may have processors of dissimilar types. Design freedom can lead to heterogeneity, as machines can have distinct instruction sets, byte orderings, processor speeds, operating systems, and devices. Heterogeneity is a result of design autonomy, but is significant enough to deserve special mention. Because the individual processors within the system can have disparate architectures, inter-processor communication may require translation of data into a format understood by the recipient. Also, a program compiled for one architecture cannot be directly executed on a machine of another architecture. Section 2.1.1 describes heterogeneity issues in greater detail.

communication autonomy

Separate systems can make independent decisions about what information to release, what messages to send, and when to send them. A system is not required to advertise all its available facilities, nor is it required to respond to messages received from other systems. A system is free to request scheduling for a task, regardless of whether that task could be run locally.

administrative autonomy

Each system sets its own resource allocation policies, independent of the policies of other systems. The local policy decides what resources are to be shared. In particular, a local system can run in a manner counterproductive to a global scheduling algorithm. All policy-tuning parameters are set by the local administrator. Also, because membership in the system is dynamic, a machine can attempt to join any system; conversely, the module managing the administrative aspects of a system can refuse any such attempt by any machine.

execution autonomy

Each system decides whether it will honor a request to execute a task and has the right to stop executing a task it had previously accepted.

Execution autonomy allows a system to have a local scheduling policy; administrative autonomy allows the system to choose that policy. Many existing mechanisms exhibit execution autonomy but have a uniform scheduling policy for all participating machines, and thus do not have administrative autonomy.

To be considered autonomous, a system must display some degree of all four types of autonomy. Mechanisms supporting task placement must support all four types of autonomy. Therefore, the mechanisms must run on multiple architectures, allow local decisions regarding communication with external systems and execution of tasks, and support a local scheduling policy. Unless noted otherwise, all uses of the terms *autonomous system* and *system* in this dissertation refer to autonomous, heterogeneous systems.

Because of execution and communication autonomy, all decisions pertaining to a system are under its control. The system advertises as little or as much of its system state as its local policy decrees, and cannot be forced to accept tasks for execution. Therefore, a machine A may not receive complete information describing machine B; A knows only what B chooses to tell A.

The execution autonomy constraint requires a system to be able to suspend a task and remove it from a processor if the local scheduling policy determines that it should no longer be run. Removal of a task is called *task revocation*. Revocation can be accomplished by killing the task, by suspending the task, or by moving it to a new processor (this is called *task migration*).

The combination of communication and design autonomy, and execution autonomy poses another problem for process migration. Execution autonomy can require the migration mechanisms to move a process from one machine to another, but because of communication autonomy and design autonomy, the sender may not know the architecture of the recipient machine. Therefore, advance translation of the program image might be impossible. Section 2.1.1 discusses related work on machine-independent program representation that could alleviate this problem. Machines with different instruction sets cannot directly share code. The mechanisms presented in this dissertation provide support for, but do not include, migration of architecture-dependent processes between heterogeneous systems.

Administrative autonomy means that a system cannot rely on neighboring systems to behave in any specific manner. When combined with communication autonomy, it means that expected inter-message times may be nonuniform between neighbors, because they may not send messages with the same frequency. The combination of administrative autonomy and execution autonomy means that the local scheduling

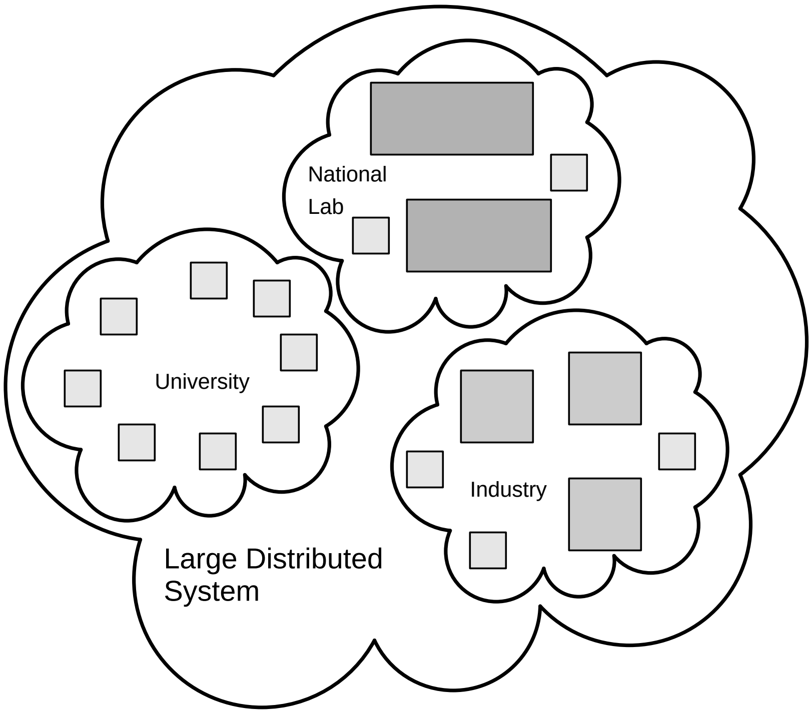

Figure 1.1 A sample distributed system

policy might act contrary to the concerted efforts of a group of cooperating remote modules.

The mechanisms described in this dissertation support global task scheduling in autonomous, heterogeneous, distributed systems. Figure 1.1 depicts a sample system representing a cooperative effort among a university, a federally-funded national laboratory, and a private corporation. Each organization contributes some of its computational resources (workstations from the university, mainframes from industry, and supercomputers from the national lab), and the resulting system provides a variety of computational resources with more aggregate power than any single organization possesses. At the same time, each individual organization preserves some of its autonomy and reserves the right to decide what runs on its machines.

We draw a distinction between the scheduling support mechanisms and the scheduling policies and associated algorithms built upon these mechanisms. The algorithms that implement the policies are responsible for deciding where a task should be run;

the mechanisms are responsible for gathering the information required by the algorithms and for carrying out the policy decisions. The mechanisms provide capability; the policies define how that capability is to be used. An *administrator* is an entity, either a human or a software module, that decides the policy for a system.

1.1 Statement of Thesis

THESIS

> *Automated support for the placement of tasks in distributed, autonomous, heterogeneous systems can be achieved in a scalable manner while preserving autonomy and supporting a variety of scheduling algorithms.*

Five principles guided the development of the mechanisms described by the thesis statement. Each principle addresses part of the thesis statement, and together they form a basis for constructing scheduling support mechanisms that fulfill the thesis.

generality The support mechanisms should support a broad spectrum of algorithms, and should be extensible to support current and future scheduling policies. In particular, the representations of system capabilities and task requirements should adapt to the needs of users and administrators.

autonomy There should be as little forfeiture of local control as is feasible. The mechanisms should support the autonomy of the policy for each system; only those data the local policy wishes to advertise to other systems should be advertised. Each machine within the system should be free to have a local scheduling policy that does not conform to a global policy. Parameters that control the system's behavior should be tunable by the local administrator.

scalability The support mechanisms should function on systems ranging from a single workstation to hundreds or thousands of processors, with interconnection schemes ranging from local area networks to wide area networks. Centralization of information precludes scalability (see [Stu88]), and should therefore be avoided.

non-interference High monitoring overhead and message traffic can adversely affect performance within the system. Therefore, the scheduling mechanisms should communicate only necessary data to minimize their interference with the running of application programs. The scheduling module should also minimize the use of memory, disk and other shared resources.

data soundness An ideal support mechanism supplies complete, perfectly accurate information to scheduling algorithms. Lamport [Lam78] discusses information dissemination latency in distributed systems, and shows that it is impossible to know the state of the entire distributed system instantaneously. The best that can be achieved is an estimate of the state at some point in the past. However, the underlying mechanisms can guarantee individual properties of the data. Of primary concern are the timeliness, completeness, and accuracy of the data available to the algorithms.

Conflicts can arise when two or more of the principles are observed to their fullest extent. For example, the distributed and autonomous nature of the system precludes global sharing of information. There is an obvious tradeoff between freshness of data and minimization of resources spent collecting the data. Frequent and detailed updates support data soundness but increase overhead. The support mechanisms in this dissertation provide facilities to tune system behavior, giving administrators freedom to choose from a range of performance alternatives.

Because of scalability and distribution, no machine can keep complete information on every processor in a large system. The bookkeeping requirements are proportional to the number of machines in the distributed system, the set of information describing each machine, and the amount of communication each machine does. This bookkeeping could quickly consume the processing power of the system, and little or no productive work would be accomplished (see [WV80, FR90]). Again, there is a tradeoff between the accuracy of a system description and its size. A solution is to compress multiple system descriptions into one, thus saving space while preserving much of the descriptive information.

Assumptions about behavior of the computing systems that make up the distributed system violate the autonomy principle. With complete autonomy, it is unlikely that any useful work would be accomplished in a distributed system, because there is no assurance that any of the individual systems are conforming to any standard of behavior. Tradeoffs between autonomy and the other principles occur often, and are resolved in favor of autonomy to the greatest extent possible while still fulfilling the basic requirements of scheduling support.

Therefore, when a conflict between autonomy and one of the other design principles occurs, the resulting solution is formulated to include as little mechanism as necessary and sacrifice the least autonomy that will resolve the conflict. A discussion of these conflicts and compromise solutions appear in chapter 3.

1.2 Organization of the Dissertation

The remainder of this dissertation is organized as follows: chapter 2 discusses related work; chapter 3 describes an architecture for distributed systems; chapter 4 defines formal system state dissemination rules and proves them correct; chapter 5 describes the MESSIAHS prototype implementation of these mechanisms; chapter 6 gives performance results; and chapter 7 draws conclusions and states future directions for this work.

2. RELATED WORK

This chapter describes three areas of related work. The first section discusses existing mechanisms that underlie the work in this dissertation, including support for heterogeneity and task migration. Heterogeneous support is necessary for interoperability between machines with design autonomy. Task migration supports execution autonomy. The second section describes related work on mechanisms supporting scheduling for distributed systems. Section two also examines the systems in light of the five principles given in section 1.1. The third and fourth sections examine scheduling algorithms. Section three presents a taxonomy of scheduling algorithms for distributed systems, and section four relates a survey of distributed scheduling algorithms and systems.

2.1 Extant Fundamental Mechanism

The mechanisms described in this dissertation are built upon fundamental mechanisms developed by others. These extant components consist of support for heterogeneous computing and task migration mechanisms. Heterogeneous computing support, in the form of a uniform data representation and an architecture-independent program representation, is necessary to support design autonomy fully. Task migration provides options for execution autonomy beyond task suspension or termination.

These issues are described here because they support important aspects of autonomy. The mechanisms developed as part of the thesis research for this dissertation take advantage of this existing work and do not include explicit mechanisms that duplicate existing functionality. The mechanisms were designed with these needs in mind, and nothing in the design precludes the use of extant techniques; rather, the design assumes that these techniques are available, and can use them where they are available.

2.1.1 Heterogeneous Computing

Machines with heterogeneous architectures often possess differing data representations. For two heterogeneous machines to communicate effectively, each must be able to translate data to a format understood by the other. Machines with a common data representation can communicate directly without resorting to an external data representation.

Architectural heterogeneity of data representation is usually accommodated by one of two approaches for communication among machines with n disparate architectures: either every machine has $n-1$ conversion modules, one to convert from its local data encoding to the encoding for each of the other $n-1$ architectures in use; or all machines can have a module to encode and decode from their architecture's representation into a standard, common format. The former approach, called *asymmetric conversion*, requires $O(n^2)$ different conversion modules, while the latter approach, called *symmetric conversion*, requires $O(n)$ distinct modules (see Comer and Stevens [CS93b, chapter 19]). The approach of symmetric data conversion is generally preferred because of the relative ease of adding new data formats to the distributed system.

ISO X.409, Abstract Syntax Notation One (ASN.1) is the international standard for external data representation [fS87a, fS87b], and specifies an explicit encoding. Explicit encoding embeds type information in the data stream, and a host with no prior knowledge of the data structure can interpret the data. The XDR (eXternal Data Representation) standard [Sun87] specifies an implicit encoding for data types, which means that no type information is embedded in the data stream. The hosts at the endpoints of a communication must agree upon the structure of the data beforehand.

A situation analogous to the external data representation problem exists for the representation of compiled programs for heterogeneous systems. As of yet, no one has specified an architecture-independent and operating-system-independent program representation. Machines that share a common object code format and instruction set can share object files without translation.

Various attempts have been made that have resulted in $O(n^2)$ solutions to the problem. Essick [EI87], and Shub, et al. [DRS89, Shu90] devise multiple-architecture task representations. Both representations combine machine code for multiple architectures in a single program.

External program representations, analogous to external data representations, have been proposed. The Open Software Foundation has proposed ANDF (Architecture Neutral Distribution Format [Mac93]) and an associated implementation technology, TDF (Ten15 Distribution Format [Pee92]), as standards for intermediate program representation for the OSF/1 operating system. UNCOL [Con58] is an earlier effort at such a standard. While each of these addresses some aspects of supporting architecture-independent program representation, none of them is wholly satisfactory. The specification of a unified, external program representation is an open problem.

The mechanisms described by this dissertation use the existing solutions to architecture-independent data representation. The problem of determining an architecture-independent program representation is an active research area. Current research focuses on specifying intermediate forms for program compilation. Rather than preclude support for heterogeneous systems, the mechanisms described in this dissertation are designed to take advantage of such advances when they become available.

2.1.2 Task Migration

A vital component of execution autonomy is the ability to revoke a running task and to reclaim the resources used by the task. Terminating the task provides the required functionality, but to users of the distributed system, this action appears capricious and unsatisfactory. A solution to this problem, called *task migration*, moves a running task from a *source* processor to a *destination* processor.

The design of the mechanisms discussed in chapter 3 assumes the presence of an underlying checkpointing and task-migration mechanism for the support of task revocation and execution autonomy. This section gives a brief survey of several alternatives for process migration mechanisms. It first examines one in detail, and then presents several mechanisms as examples, noting the unique features of each.

Task migration mechanisms use variations on a three-step process:

1. (checkpoint) The source system stops a running process and saves its state.

2. (transfer) The saved state of the process is transferred to the destination system, and resources are released at the source system.

3. (restart) The destination system restarts the process.

If the process migration mechanism is *transparent*, the process will not detect that it has moved. There are several factors that complicate the mechanism. For example, open files and communication endpoints must be replicated on the destination system to achieve transparent migration. In general, any location-dependent aspects of the process impede migration.

DEMOS/MP

Powell and Miller [PM83] discusses process migration in the DEMOS/MP distributed operating system. DEMOS/MP is a message-based operating system, and all interactions between processes occur via communications-based system calls. DEMOS/MP splits the transfer step into six substeps, yielding an eight-step migration mechanism. The steps follow, with actions by the source and destination machines marked.

1. (source) Stop the executing process, and mark it in migration.

2a. (source) Request migration by the destination.

2b. (destination) Allocate a new process state within the kernel.

2c. (destination) Copy the process state from the source.

2d. (destination) Copy the process memory (code, data, stack) from the source.

2e. (source) Forward any messages that arrived for the process during the previous steps.

2f. (source) Reclaim the resources used by the process, but keep a *forwarding address* so that messages will be correctly delivered to the destination processor.

3. (destination) Restart the process.

The numbering preserves that of the basic algorithm. Steps 2a through 2e indicate substeps of the transfer process. The DEMOS/MP approach is typical of migration mechanisms that attempt to optimize various aspects of the transfer step.

Sprite

The Sprite operating system achieves transparent process migration [OCD+88, DO91]. Sprite uses the basic checkpoint–transfer–restart algorithm, but simplifies the transfer process because of the use of *backing files*. Instead of paging to local storage, Sprite pages to ordinary files in the network file system. Thus, any machine in the Sprite system can access the backing file for a process. To implement the transfer step, the source pages out the running process and passes information describing the backing file to the destination, which uses the same file and pages in the migrated process.

The V System

The V system uses a technique called *precopying*, wherein the memory is copied while the process continues to execute [TLC85]. After the memory is precopied, the process is stopped and any altered pages are copied again. This reduces the amount of time a process is frozen.

Accent

The Accent operating system uses a *lazy* approach to transfer [Zay87]. The virtual memory image of the process is left on the source, and as page faults occur on the destination, the memory is moved one page at a time. Lazy copying has the advantage that unneeded memory is never copied, but the disadvantage that resources cannot be immediately reclaimed at the source.

Locus, Charlotte, and work by Bryant and Finkel

Locus uses the basic checkpoint–transfer–restart algorithm, with the optimization that read-only segments that already exist on the target machine are not copied [PW85]. The Charlotte distributed operating system uses the basic algorithm, with the addition of message endpoint forwarding [AF87, FA89]. Bryant and Finkel [BF81] concentrates on developing stable process migration methods. A stable method avoids process thrashing, which occurs when the migration of a task immediately induces another migration.

2.2 Scheduling Support Systems

Solutions to the problem of scheduling support for distributed systems have been proposed, but none of the proposals fulfill all the goals set forth in the thesis statement in section 1.1. This section describes prior research in scheduling support mechanisms. In this discussion, the terms *local task* and *foreign task* are defined from the point of view of the host executing the task. A local task executes on the host where it originated, without going through the global scheduling system. A foreign task originates at a host different from the one on which it executes.

NetShare

NetShare is a distributed systems construction product of Aggregate Computing, Inc. [Agg93c, Agg93b, Agg93a]. NetShare comprises services that provide resource management and task execution on a heterogeneous local-area network. NetShare has two main components, the Resource Management Subsystem and the Task Management Subsystem.

The Resource Manager consists of three parts: the Resource Information Server (RIS), the Resource Agents (RA), and the Client Side Resource Library (CSRL). The RIS is a centralized database of information describing resources available within the system, including state information for individual machines. Resource Agents run on each machine and advertise their system state to the RIS. Clients use the CSRL to request resource allocation through the RIS. The CSRL is a library of function calls that are linked with individual application programs. There is no scheduling agent external to the applications; they are self-scheduling.

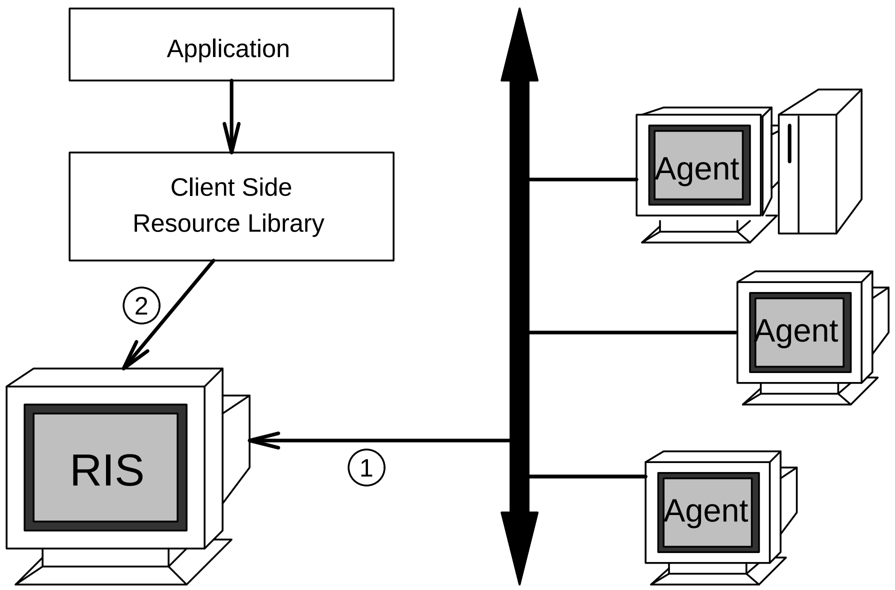

Figure 2.1 The NetShare Resource Management Subsystem

Figure 2.1 shows the interaction between an application, the RIS, and Resource Agents. In step (1), state information passes from Agents to the RIS. In step (2), the application uses the CSRL to query the RIS.

The Agent updates consist of the following information:

- the name, architecture, model, and network address of the host
- the name, version, and release of the operating system
- the amount of physical, free virtual, and used virtual memory
- one, five, and 15 minute load averages
- the idle time and CPU usage of the host
- power rating, based on standard benchmarks
- number of users
- two user-definable properties

The two user-definable properties provide a limited extension mechanism for NetShare. Clients query the database through the CSRL, and receive a set of matching records in response. A sample call to the CSRL, which appears in [Agg93c], is:

```
select UNIX_HOST if ((UNIX_HOST:LOAD_5 < 1.0) &&
                     (UNIX_HOST:USERS == 0))
         order by (UNIX_HOST:LOAD_5)
```

This call queries the database for hosts running the UNIX operating system, with a five-minute load average less than 1.0, and no active users. The RIS finds the matching set of hosts, and returns the set, sorted by five-minute load average. The syntax and use of the resource management mechanism is similar to that found in the Univers [BPY90] and Profile [Pet88] naming systems.

The client uses the Task Management Subsystem (TMS) to schedule the individual tasks for execution. The TMS is composed of the Task Servers (TS) and the Client Side Task Library (CSTL). Application programs place individual tasks with calls to the CSTL.

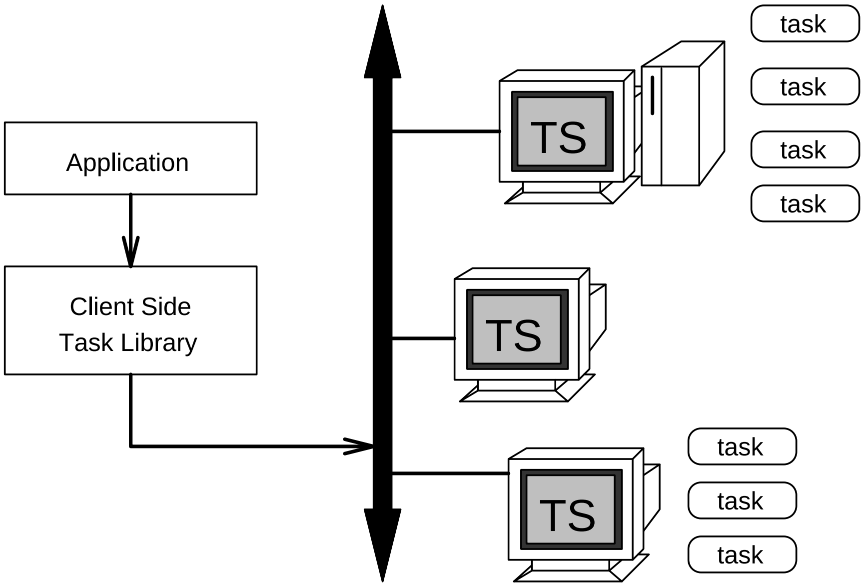

Figure 2.2 The NetShare Task Management Subsystem

Figure 2.2 shows the relationship between the application and the Task Servers. In the depicted scenario, the client application has selected two servers using the RMS, and has used the TMS to place seven tasks on the servers.

Task Servers have limited support for autonomy, in that administrators can set quotas limiting the number of tasks that are either placed by a host (an *export quota*), or that have been accepted from a foreign host (*import quotas*).

NetShare has several limitations that prevent it from meeting the guidelines expressed in chapter 1. NetShare uses centralized information and file storage, which limit the scalability of systems that use NetShare. The two-field extension mechanism prohibits elaborate scheduling policies or multiple policies that use data not in the standard set. Execution autonomy is compromised because the policy expression mechanism is completely under the control of the application program; the acceptance of a task for execution is based solely on the import quota of the target machine.

Load Balancer

Load Balancer [Fre], a product of Freedman Sharp and Associates Inc., is a batch queuing and load sharing system for UNIX operating systems. Load Balancer is similar to NetShare in that Load Balancer has a single resource manager (lbmasterd), a per-node local task manager (lblocald), centralized decision making, and limited autonomy support.

Load Balancer has a static configuration file covering all hosts, users, and applications within the system. The lbmasterd reads the configuration file, and is responsible for directing scheduling application programs to hosts for execution. Table 2.1 contains the fields in the configuration file describing hosts and tasks.

Table 2.1 Load Balancer configuration file fields

host description
hours of availability
cpu architecture
RAM
swap space
maximum number of foreign tasks
maximum system load
number of CPUs

application description
revocation behavior
per-architecture estimated runtime
RAM usage
swap space usage

The lblocald process collects system state information and forwards it to the lbmasterd. lbmasterd also performs local task management on tasks scheduled by Load Balancer, and determines when a system is considered idle, and thus eligible to accept Load Balancer tasks.

The revocation behavior for an application is centrally specified in the configuration file, which violates administrative autonomy for the individual hosts. There is no provision for extending the description mechanism, although the presence of default host, user, and application descriptions provides some generality. Because of

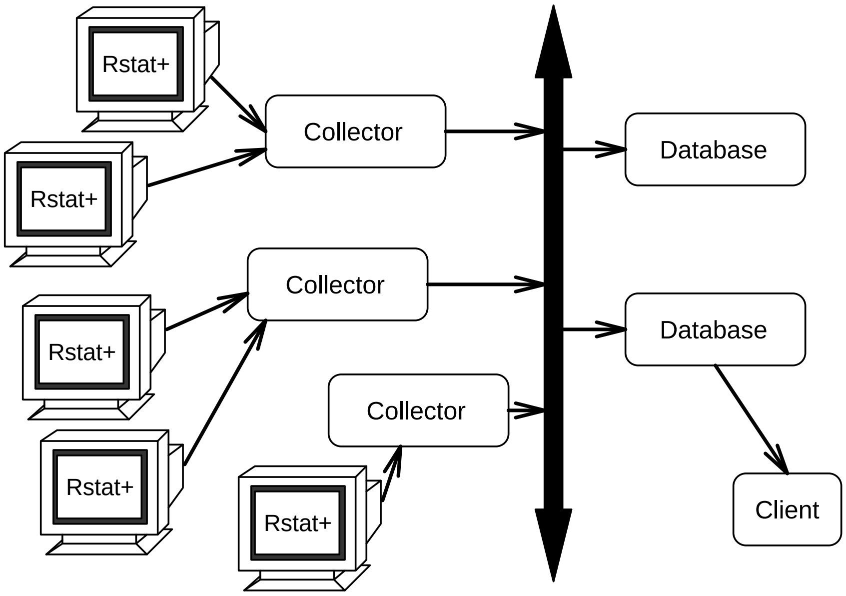

Figure 2.3 A sample DRUMS system

its centralized decision making, Load balancer violates the scalability and execution autonomy requirements.

DRUMS and ANY

DRUMS[1] [BH91a, BH91b, Bon91] is a distributed information collection and management system developed at Victoria University, Wellington. DRUMS has three main components: local system state monitors (rstat+), processes that collect information from a set of hosts (collector), and a replicated centralized information manager (database). Figure 2.3 shows the structure of a DRUMS system.

A collector process introduces one layer of hierarchical structure, and periodically queries a set of hosts to obtain their system state from the rstat+ daemons. A collector process then broadcasts the data to all the database processes. The database processes store the system description data for all hosts in the system and respond to client requests.

[1]DRUMS: Distributed Resource MeasUrement Service

DRUMS concentrates on data-collection, and uses an associated scheduler called ANY. ANY queries a database to obtain a set of hosts that match a description. A description contains a set of desired characteristics and a list of resource weightings. The characteristics are grouped as follows:

Statistical measurements, such as system load, swap space, and memory availability

Hardware requirements, such as vector processors, a floating-point processor, or local disks

Architecture and system software, including the processor type (e.g. Sun SPARC) and operating system (e.g. 4.3BSD UNIX)

User interaction, including the presence of a user on the console, and the number of users logged in to the machine

Hostnames and network addresses, which allow the restriction of queries to particular hosts or networks

These characteristics can be combined with logical AND, OR, and NOT operators. The resulting set of matching hosts is sorted based on the weightings given in the description, and the highest-ranked hosts are returned.

DRUMS and ANY provide a data collection mechanism and the ability to customize the scheduling policy of a local host. However, the centralized database processes limit scalability, particularly because of the broadcast mechanism used by collector processes. There is no provision within DRUMS for local hosts to reject foreign tasks, which precludes execution autonomy. DRUMS does not provide extensible description mechanisms, and therefore does not follow the principle of generality.

Remote UNIX, Condor, Butler, and Distributed Batch

Remote UNIX and its successor, Condor, were developed at the University of Wisconsin [Lit87, BLL92]. Butler was developed as part of the Andrew project at Carnegie-Mellon University [Nic87], and Distributed Batch was developed at the MITRE Corporation [GSS89]. All of these systems attempt to increase utilization and share load across a set of UNIX-based workstations, but are less complete systems than NetShare or Load Balancer. Therefore, this section groups these systems together and gives a brief description of each.

Remote UNIX and Condor use a Central Resource Manager, which gathers information about all the participating hosts, and a Local Scheduler per host that controls task execution for that host. Remote UNIX has a simple two-level priority scheme for local and foreign tasks, while Condor has a policy expression mechanism that provides administrative autonomy. The centralized control of the Central Resource Manager limits scalability. There is no provision for extension of the description mechanism. Both Condor and Remote UNIX support checkpointing and task migration to facilitate execution autonomy, but neither provides support for communication autonomy.

Butler uses a central Machine Registry and a shared file system to manage a set of homogeneous workstations. There is no provision for administrative or execution autonomy; all control is centralized. Hosts are dedicated to one task at a time, and are not returned to the free pool until the task completes execution.

Distributed Batch runs on a local-area network of 4.2BSD UNIX workstations, using centralized storage. Distributed Batch contains revocation support in the form of task termination, suspension, and migration. Hosts can be selected based on architecture, operating system version, available memory, local disk configuration, and floating point hardware. There is no administrative autonomy within Distributed Batch.

All of these systems violate the principle of scalability because of centralized file storage and information broadcasting. None of these systems provide support for all four aspects of autonomy. These systems do not meet the requirements of generality, in part because they have no extensible description mechanisms.

2.3 A Taxonomy of Scheduling Algorithms

Casavant and Kuhl [CK88] proposes a taxonomy for scheduling algorithms in distributed systems, which is reproduced in figure 2.4. Section 2.4 surveys related work in the area of scheduling algorithms, and classifies those algorithms in terms of this taxonomy. Chapter 6 describes experiments performed using algorithms chosen to represent different portions of the taxonomy.

At the topmost layer, schedulers are either *local* or *global.* Global scheduling, or macro-scheduling, chooses where to run a task. Local scheduling, or micro-scheduling, chooses which eligible task executes next on a particular processor. This dissertation concentrates on support for global scheduling, and uses of the term scheduling refer to global scheduling throughout the remainder of the dissertation.

Global scheduling has two subcategories: *static* and *dynamic* scheduling. Static, or compile-time, scheduling depends only on the makeup of the task force and the topology of the distributed system. Static schedulers assume that precise system and task description information is available at the time the program is compiled. Dynamic, or run-time, scheduling takes system state into account, and makes all decisions regarding the placement of a task at the time it is executed.

In *physically non-distributed* scheduling policies, a single processor makes all decisions regarding task placement. Under *physically distributed* algorithms, the logical authority for the decision-making process is distributed among the processors that constitute the system.

Under *non-cooperative* distributed scheduling policies, individual processors make scheduling choices independent of the choices made by other processors. With *cooperative* scheduling, the processors subordinate local autonomy to the achievement of a common goal.

Both static and cooperative distributed scheduling have *optimal* and *suboptimal* branches. Optimal assignments can be reached if complete information describing the system and the task force is available. Suboptimal algorithms are either *approximate*

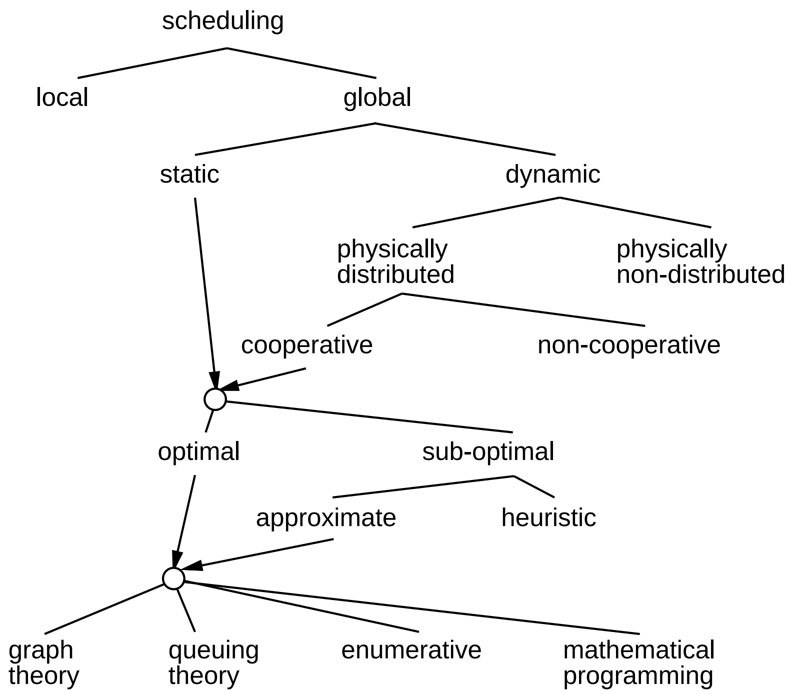

Figure 2.4 A taxonomy of scheduling algorithms from [CK88]

or *heuristic*. Heuristic algorithms use guiding principles, such as assigning tasks with heavy inter-task communication to the same processor, or placing large jobs first. Approximate solutions use the same computational methods as optimal solutions, but use solutions that are within an acceptable range, according to an algorithm-dependent metric.

Approximate and optimal algorithms employ techniques based on one of four computational approaches: enumeration of all possible solutions, graph theory, mathematical programming, or queuing theory.

There are other properties of scheduling algorithms that are not represented in the taxonomy, but apply to several different branches simultaneously. These properties are *adaptive*, *bidding*, *load balancing*, *probabilistic*, and *one-time assignment* or *dynamic reassignment*.

Any of the dynamic categories can have a subclass containing adaptive algorithms. Many researchers use the terms dynamic and adaptive interchangeably; in this dissertation, the term adaptive refers only to algorithms that employ history mechanisms to track system response to past scheduling decisions, and modify the scheduling algorithm accordingly. Bidding algorithms advertise work to be done and wait for responses from available processors.

Load balancing policies attempt to distribute the workload so that processor utilization is approximately equal for all processors in the system. Eager, et al. [ELZ85] discusses load-balancing algorithms using task migration in detail and classifies them according to whether they are *sender-initiated* or *receiver-initiated*. Under sender-initiated load balancing, the busy processor finds an idle processor to receive a task. With receiver-initiated load balancing, an idle processor locates an overloaded processor and requests a task. Wang and Morris [WM85] presents a similar taxonomy using the names source-initiated for sender-initiated and server-initiated for receiver-initiated load balancing.

Probabilistic algorithms operate in one of two methods. The first method makes scheduling choices based on statistics rather than exact information. The second method randomly orders the tasks within a task force, then schedules the tasks in that order. Algorithms using the latter method produce several such schedules, and choose the best among them, relying on the randomness of the ordering to produce at least one acceptable schedule.

With one-time assignment algorithms, a task runs for its entire lifetime on the processor where it is initially scheduled. Dynamic reassignment algorithms attempt to perform migration of tasks to more suitable processors.

The work presented in this dissertation provides support for global dynamic scheduling algorithms. Global static algorithms can be implemented using the mechanisms, but such implementations will not produce optimal results. These issues are discussed in chapter 6, Experimental Results.

2.4 Scheduling Algorithms

Many researchers have devised algorithms for task placement in distributed systems. This section categorizes several of these techniques in terms of the taxonomy presented in the previous section, and analyzes their applicability to the general problem of global scheduling for autonomous, heterogeneous, distributed systems.

Tables 2.2, 2.3, and 2.4 display information garnered from our survey of existing scheduling algorithms and mechanisms. For each algorithm or mechanism, an entry indicates whether a method is a policy, mechanism, or both; whether the method is distributed, or supports heterogeneity or autonomy; and whether the method minimizes overhead, supports scalability, or is extensible. Entries are either *Y*, *N*, *P*, or *x*, indicating the answer is yes, no, partially, or not applicable. In the case of autonomy, the letters *A*, *C*, or *E* indicate support for administrative, communications, or execution autonomy. Design autonomy is not listed, as it is covered by the heterogeneity column. The remainder of this section contains a brief description of each method, with a discussion of its place in the taxonomy and its individual properties.

2.4.1 Dynamic, Distributed, Cooperative, Suboptimal Algorithms

All of the algorithms in this section are dynamic, distributed, cooperative, suboptimal, and heuristic.

Blake [Bla92] describes four suboptimal, heuristic algorithms. Under the first algorithm, Non-Scheduling (NS), a task is run where it is submitted. The second algorithm is Random Scheduling (RS), wherein a processor is selected at random and is forced to run a task. The third algorithm is Arrival Balanced Scheduling (ABS), in which the task is assigned to the processor that will complete it first, as estimated by the scheduling host. The fourth method uses receiver-initiated load balancing, and is called End Balanced Scheduling (EBS). NS, RS, and ABS use one-time assignment; EBS uses dynamic reassignment.

Table 2.2 Summary of distributed scheduling survey, part I

Method	Policy	Mechanism	Distributed	Heterogeneous	Autonomy	Overhead	Scalable	Extensible
Blake [Bla92] (NS)	Y	N	Y	N	C	Y	Y	x
(ABS)	Y	N	Y	N	N	N	Y	x
(RS)	Y	N	Y	N	N	Y	Y	x
(EBS)	Y	N	Y	N	N	N	Y	x
(CBS)	Y	N	N	N	N	N	N	x
Condor [BLL92]	Y	Y	P	N	A	x	N	N
Remote UNIX [Lit87]	Y	Y	P	N	A	x	N	N
Butler [Nic87]	Y	Y	P	Y	A	x	N	N
MITRE [GSS89]	Y	Y	P	P	A	x	N	N
Casavant and Kuhl [CK84]	Y	Y	Y	N	E	x	P	N
Ghafoor and Ahmad [GA90]	Y	Y	Y	N	E	Y	P	N
Stankovic [Sta81, Sta85a]	Y	N	Y	N	N	x	N	x
Ramamritham and Stankovic [RS84]	Y	N	Y	N	E	x	N	x
Wave Scheduling [VW84]	Y	Y	Y	N	E	x	P	N
Ni and Abani [NA81] (LED)	Y	N	Y	N	N	x	N	N
(SQ)	Y	N	Y	N	N	Y	Y	x

Table 2.3 Summary of distributed scheduling survey, part II

Method	Policy	Mechanism	Distributed	Heterogeneous	Autonomy	Overhead	Scalable	Extensible
Stankovic and Sidhu [SS84]	Y	Y	Y	N	EAC	x	P	N
Stankovic [Sta85b]	Y	N	Y	N	N	x	N	N
Andrews et al. [ADD82]	Y	N	Y	x	E	x	Y	N
Majumdar and Green [MG80]	Y	Y	Y	N	N	x	N	N
Bonomi [Bon90]	Y	N	N	N	N	x	N	N
Bonomi and Kumar [BK90]	Y	N	N	Y	N	x	N	N
Greedy Load-Sharing [Cho90]	Y	N	Y	N	N	X	Y	N
Gao, et al. [GLR84] (BAR)	Y	N	Y	N	N	x	N	N
(BUW)	Y	N	Y	N	N	x	N	N
Stankovic [Sta84]	Y	N	Y	N	N	x	P	N
Chou and Abraham [CA83]	Y	N	Y	N	N	x	Y	N
Bryant and Finkel [BF81]	Y	N	Y	N	N	x	Y	N
Chow and Kohler [CK79]	Y	N	N	Y	N	x	N	N
Casey [Cas81] (dipstick)	Y	N	Y	N	E	x	N	N
(bidding)	Y	N	Y	N	E	x	N	N
(adaptive learning)	Y	N	Y	N	N	Y	Y	N

Table 2.4 Summary of distributed scheduling survey, part III

Method	Policy	Mechanism	Distributed	Heterogeneous	Autonomy	Overhead	Scalable	Extensible
Hwang et al. [HCG+82]	Y	Y	Y	Y	N	x	N	N
MICROS [WV80]	Y	Y	Y	N	N	Y	Y	N
Klappholz and Park [KP84] (DRS)	Y	N	Y	N	N	x	Y	N
Reif and Spirakis [RS82]	Y	N	Y	N	N	x	N	N
Ousterhout, et al. [OSS80]	Y	Y	N	N	N	x	N	N
Bergmann and Jagadeesh [BJ91]	Y	N	N	N	N	x	N	N
Drexl [Dre90]	Y	N	N	Y	N	x	x	N
Hochbaum and Shmoys [HS88]	Y	N	N	Y	N	x	x	N
Hsu, et al. [HWK89]	Y	N	N	Y	N	x	x	N
Stone [Sto77]	Y	N	N	Y	N	x	x	N
Lo [Lo88]	Y	N	N	Y	N	x	x	N
Price and Salama [PS90]	Y	N	N	Y	N	x	x	N
Ramakrishnan et al. [RCD91]	Y	N	N	Y	N	x	x	N
Sarkar [Sar89]	Y	N	N	Y	N	x	x	N
Sarkar and Hennessey [SH86b]	Y	N	N	Y	N	x	x	N

Casavant and Kuhl [CK84] describes a distributed task execution environment for UNIX System 7, with the primary goal of load balancing without altering the user interface to the operating system. As such, the system combines mechanism and policy. This system supports execution autonomy, but not communication autonomy or administrative autonomy.

Ghafoor and Ahmad [GA90] describes a bidding system that combines mechanism and policy. A module called an Information Collector/Dispatcher runs on each node and monitors the local load and that of the node's neighbors. The system passes a task between nodes until either a node accepts the task or the task reaches its transfer limit, in which case the current node accepts the task. This algorithm assumes homogeneous processors and has limited support for execution autonomy.

Stankovic [Sta81, Sta85a] describe methods for homogeneous systems based on Bayesian decision theory. There is no support for autonomy, nor are the methods scalable because they require full knowledge of all nodes in the system. Ramamritham and Stankovic [RS84] presents a distributed scheduling algorithm for hard real-time systems. This work supports a form of execution autonomy that guarantees a hard real-time deadline. A node can choose to accept a task and guarantee its completion by a deadline, or to decline the task.

Van Tilborg and Wittie [VW84] presents Wave Scheduling for hierarchical virtual machines. The task force is recursively subdivided and the processing flows through the virtual machine like a wave, hence the name. Wave Scheduling combines a non-extensible mechanism with policy, and assumes the processors are homogeneous.

Ni and Abani [NA81] presents two dynamic methods for load balancing on systems connected by local area networks: Least Expected Delay and Shortest Queue. Least Expected Delay assigns the task to the host with the smallest expected completion time, as estimated from data describing the task and the processors. Shortest Queue assigns the task to the host with the fewest number of waiting jobs. These two methods are not scalable because they use information broadcasting to ensure complete information at all nodes. [NA81] also presents an optimal stochastic strategy using mathematical programming.

The method described in Stankovic and Sidhu [SS84] uses task clusters and distributed groups. Task clusters are sets of tasks with heavy inter-task communication that should be on the same host. Distributed groups also have inter-task communication, but execute faster when spread across separate hosts. This method is a bidding strategy, and uses non-extensible system and task description messages.

Stankovic [Sta85b] lists two scheduling methods. The first is adaptive with dynamic reassignment, and is based on broadcast messages and stochastic learning automata. This method uses a system of rewards and penalties as a feedback mechanism to tune the policy. The second method uses bidding and one-time assignment in a real-time environment, similar to that in [SS84].

Andrews, et al. [ADD82] describes a bidding method with dynamic reassignment based on three types of servers: free, preferred, and retentive. Free server allocation will choose any available server from an identical pool. Preferred server allocation asks for a server with a particular characteristic, but will take any server if none is available with the characteristic. Retentive server allocation asks for particular characteristics, and if no matching server is found, a server, busy or free, must fulfill the request.

Majumdar and Green [MG80] discusses the Real Time Resource Manager, a load-balancing system running on multiple VAX 11/780 computers. A module runs on each participating system, with five functional components: DETECT, which checks for an overload on the local processor; STATUS, which generates a status report in response to an overload detection on another processor; PRESCHED, which preschedules the reconfiguration task; RECONF, which reconfigures the system; and REINIT, which reinitializes the system after a reconfiguration. RTRM uses broadcast communication of a non-extensible description, which limits scalability, and does not support autonomy.

Bonomi [Bon90] discusses properties of the Join the Shortest Queue (JSQ) heuristic for load balancing, and presents a heuristic that performs better, based on queuing theory. Bonomi and Kumar [BK90] presents an adaptive heuristic based on stochastic splitting of task forces and shows that in a least-squares sense, the heuristic balances the server idle times.

Chowdhury [Cho90] describes the Greedy load-sharing algorithm. The Greedy algorithm uses system load to decide where a job should be placed. This algorithm is non-cooperative in the sense that decisions are made for the local good, but it is cooperative because scheduling assignments are always accepted and all systems are working towards a global load balancing policy.

Gao, et al. [GLR84] describes two load-balancing algorithms using broadcast information. The first algorithm balances arrival rates, with the assumption that all jobs take the same time. The second algorithm balances unfinished work. Stankovic [Sta84] gives three variants of load-balancing algorithms based on point-to-point communication that compare the local load to the load on remote processors. Chou and Abraham [CA83] describes a class of load-redistribution algorithms for processor-failure recovery in distributed systems.

The work presented in Bryant and Finkel [BF81] combines load balancing, dynamic reassignment, and probabilistic scheduling to ensure stability under task migration. This method uses neighbor-to-neighbor communication and forced acceptance to load balance between pairs of machines. Chow and Kohler [CK79] presents load-balancing strategies using a centralized job controller, based on analysis of queuing theory models of heterogeneous distributed systems.

Casey [Cas81] gives an earlier and less complete version of the Casavant and Kuhl taxonomy, with the term *centralised* replacing *non-distributed* and *decentralised* substituting for *distributed*. This paper also lists three methods for load balancing: Dipstick, Bidding, and Adaptive Learning, then describes a load-balancing system whereby each processor includes a two-byte status update with each message sent. The Dipstick method is the same as the traditional watermark processing found in many operating systems [Com84]. The Adaptive Learning algorithm uses a feedback mechanism based on the run queue length at each processor.

Hwang, et al. [HCG+82] describes a specialized implementation of a distributed UNIX project for a network of Digital Equipment Corporation machines, and an associated load-balancing strategy. This project supports neither autonomy nor scalability.

Wittie and Van Tilborg [WV80] describes MICROS and MICRONET. MICROS is the load-balancing operating system for MICRONET, which is a reconfigurable and extensible network of 16 LSI-11 nodes. MICROS uses hierarchical structuring and data summaries within a tree structured system. All scheduling takes place in a master/slave relationship, so autonomy is not supported.

2.4.2 Dynamic Non-cooperative Algorithms

Klappholz and Park [KP84] describes Deliberate Random Scheduling (DRS) as a probabilistic, one-time assignment method to accomplish load balancing in heavily-loaded systems. Under DRS, when a task is spawned, a processor is randomly selected from the set of ready processors, and the task is assigned to the selected processor. DRS dictates a priority scheme for time-slicing, and is thus a mixture of local and global scheduling. There is no administrative autonomy or execution autonomy with this system, because DRS is intended for tightly-coupled machines.

Reif and Spirakis [RS82] presents a Resource Granting System (RGS) based on probabilities and using broadcast communication. This work assumes the existence of either an underlying handshaking mechanism or of shared variables to negotiate task placement. The use of broadcast communication to keep all resource providers updated with the status of computations in progress limits the scalability of this algorithm.

2.4.3 Dynamic Non-distributed Algorithms

Ousterhout, et al. [OSS80] describes Medusa, a distributed operating system for the Cm* multiprocessor. Medusa uses static assignment and centralized decision making it a combined policy and mechanism. It does not support autonomy, nor is the mechanism scalable.

In addition to the four distributed algorithms already mentioned, Blake [Bla92] describes a fifth method called Continual Balanced Scheduling (CBS), that uses a centralized scheduler. Each time a task arrives, CBS generates a mapping within two time quanta of the optimum, and causes tasks to be migrated accordingly. The centralized scheduler limits the scalability of this approach.

2.4.4 Static Algorithms

All the algorithms in this section are static, and as such, are centralized and without support for autonomy.

Bergmann and Jagadeesh [BJ91] describes a simple centralized scheme using a heuristic approach to schedule a task force on a set of homogeneous processors. The processors are tightly-coupled and have shared memory. The algorithm generates an initial mapping, then uses a bounded probabilistic approach to move towards the optimal solution.

Drexl [Dre90] describes a stochastic scheduling algorithm for heterogeneous systems. The algorithm uses one-time assignment, and uses a probability-based penalty function to produce schedules within an acceptable range.

Hochbaum and Shmoys [HS88] describes a polynomial-time, approximate, enumerative scheduling technique for processors with different processing speeds, called the dual-approximation algorithm. The algorithm solves a relaxed form of the bin-packing problem to produce a schedule within a parameterized factor, ϵ, of optimal. That is, the total run time is bounded by $(1 + \epsilon)$ times the optimal run time.

Hsu, et al. [HWK89] describes an approximation technique called the critical sink underestimate method. The task force is represented as a directed acyclic graph, with vertices representing tasks and edges representing execution dependencies. If an edge (α, β) appears in the graph, then α must execute before β. A node with no incoming edges is called a *source*, and a node with no outgoing edges is a *sink*. When the last task represented by a sink finishes, the computation is complete; this last task is called the critical sink. The mapping is derived through an enumerative state space search with pruning, which results in an underestimate of the running time for a partially mapped computation, and hence, the name critical sink underestimate.

Stone [Sto77] describes a method for optimal assignment on a two-processor system based on a Max Flow/Min Cut algorithm for sources and sinks in a weighted directed graph. A maximum flow is one that moves the maximum quantity of goods along the edges from sources to sinks. A minimum cutset for a network is the set of edges with the smallest combined weighting, which, when removed from the graph, disconnects all sources from all sinks. The algorithm relates task assignment to commodity flows in networks, and shows that deriving a Max Flow/Min Cut provides an optimal mapping.

Lo [Lo88] describes a method based on Stone's Max Flow/Min Cut algorithm for scheduling in heterogeneous systems. This method utilizes a set of heuristics to map from a general system representation to a two-processor system so that Stone's work applies.

Price and Salama [PS90] describes three heuristics for assigning precedence-constrained tasks to a network of identical processors. With the first heuristic, the tasks are sorted in increasing order of communication, and then are iteratively assigned so as to minimize total communication time. The second heuristic creates pairs of tasks that communicate, sorts the pairs in decreasing order of communication, then groups the pairs into clusters. The third method, simulated annealing, starts with a mapping and uses probability-based functions to move towards an optimal mapping.

Ramakrishnan, et al. [RCD91] presents a refinement of the A* algorithm[2] that can be used either to find optimal mappings or to find approximate mappings. The

[2] See Nilson [Nil80, chapter 2].

algorithm uses several heuristics based on the sum of communication costs for a task, the task's estimated mean processing cost, a combination of communication costs and mean processing cost, and the difference between the minimum and maximum processing costs for a task. The algorithm also uses ϵ-relaxation similar to the dual-approximation algorithm of Hochbaum and Shmoys [HS88].

Sarkar [Sar89] and Sarkar and Hennessey [SH86b] describe the GR graph representation and static partitioning and scheduling algorithms for single-assignment programs based on the SISAL language. In GR, nodes represent tasks and edges represent communication. The algorithm consists of four steps: cost assignment, graph expansion, internalization, and processor assignment. The cost assignment step estimates the execution cost of nodes within the graph, and communication costs of edges. The graph expansion step expands complex nodes, e.g. loops, to ensure that sufficient parallelism exists in the graph to keep all processors busy. The internalization step performs clustering on the tasks, and the processor assignment phase assigns clusters to processors so as to minimize the parallel execution time.

2.5 Summary

This chapter examined three areas of prior work related to the thesis: support for heterogeneity, extant scheduling support mechanisms, and a taxonomy and survey of existing scheduling algorithms. The mechanisms developed in later chapters assume the existence of external data representation and task migration mechanisms to support autonomy, and can take advantage of existing functionality to accomplish these tasks.

Several software systems solve restricted cases of the general problem of support for distributed scheduling, including NetShare, Load Balancer, Condor, Butler, Remote UNIX, and Distributed Batch. However, none of these meet all the requirements for generality, scalability, autonomy, data soundness, and non-interference set forth in section 1.1.

A taxonomy of scheduling policies, originally proposed by Casavant and Kuhl [CK88], was applied to a broad spectrum of scheduling algorithms from the literature. These surveyed algorithms were also analyzed with respect to the the design principles set forth in chapter 1.1 to determine what capabilities the algorithms require of the underlying mechanism.

3. DISTRIBUTED SYSTEM ARCHITECTURE

This chapter describes a model for the architecture of distributed, autonomous, heterogeneous systems. It describes the administrative and communication structure for such systems, and examines the role of individual nodes within the larger system.

3.1 The Architectural Model

This dissertation considers mechanisms supporting task scheduling in distributed systems. The architectural model for these mechanisms is hierarchical, based on observations of administrative domains in existing computer systems. A *virtual system* represents a subset of the resources of one or more real systems, and has a hierarchical structure modeling the administrative hierarchies of computer systems and institutional organization. Virtual systems can be combined into encapsulating virtual systems. For example, in figure 1.1, the University, National Lab, and Industry are each virtual systems, and are collected into a single large distributed system. Within the University, National Lab, and Industry virtual systems are other virtual systems, giving a hierarchical structure. At the lowest level of grouping, each virtual system typically consists of a subset of the capabilities of a single machine.

In this way, virtual systems combine aspects of multicomputers [Spa86] and virtual machines (see [MS70, SM79], which describe the IBM CP/67 and VM/370 operating systems). Virtual machines present the user with a subset of the capabilities of the physical machine. Multicomputers represent the capabilities of multiple machines as a single collected virtual computer. Both virtual machines and multicomputers incorporate the central concept of a virtual representation of computing resources, which is also present in virtual systems.

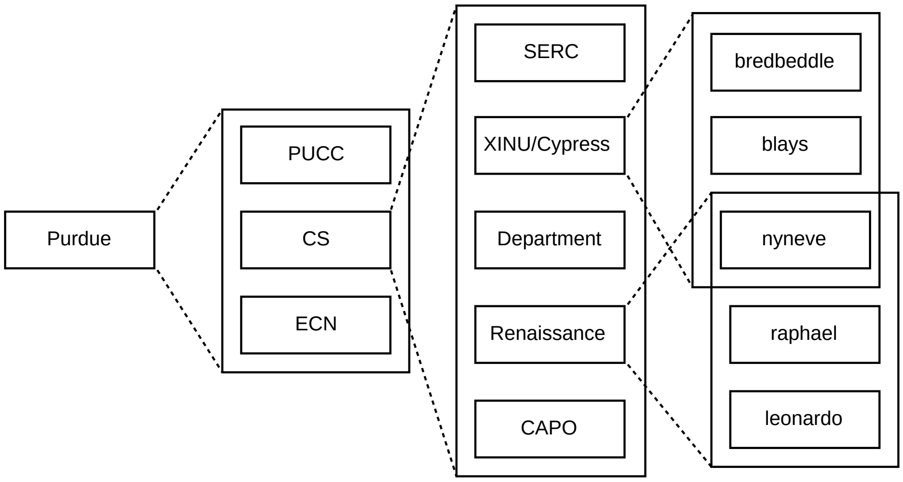

Figure 3.1 A sample virtual system

For example, the set of computers at Purdue University forms a virtual system. Within the Purdue hierarchy, subordinate virtual systems are administered by the School of Engineering (ECN), the Department of Computer Sciences (CS), and the Computing Center (PUCC), and others. The computer science machines comprise several groups: those owned by the department at large, the Software Engineering Research Center, the XINU/Cypress project, and the Renaissance project, among others. Figure 3.1 depicts this virtual system. The Renaissance system is both an encapsulating system and a subordinate. Renaissance encapsulates leonardo, raphael, and nyneve; at the same time, Renaissance is subordinate to CS.

Networks do *not* form the basis for virtual systems; administrative domains do. Virtual systems are logical, administrative groupings that may or may not correspond to physical groupings of machines. The interconnection network for a set of machines may suggest an efficient grouping of virtual systems, but it does not define the system. bredbeddle and blays are machines on the same local-area network, and owned by the same researcher, so it is natural to place them within the same virtual system. nyneve is under administrative control of two research projects, the XINU project and the Renaissance project, and therefore belongs to two virtual systems.

This hierarchical structure is similar to that presented in Feitelson and Rudolph [FR90], which describes Distributed Hierarchical Control. Under Distributed Hierarchical Control, the system uses a hierarchically-structured multiprocessor as the master processing element in a larger multiprocessor. In this system, lower levels in the control tree pass condensed and abstracted description information up to higher levels, where scheduling decisions are made (see also [WV80]).

3.1.1 Representation and System Structure

Directed acyclic graphs (DAGs) can represent virtual systems. The graph for figure 3.1 is in figure 3.2. Each vertex, or *node*, within the graph marks the root of an administrative hierarchy, and appears as a virtual system to nodes outside that administrative domain. Nodes map to physical *machines*, or *hosts*. A single node can map to multiple machines, and more than one node can map onto an individual machine.

The real capabilities of the virtual system are bounded by the combined capabilities of all its encapsulated systems, but virtual systems can advertise capabilities greater than or different from those that they actually have. The mechanisms cannot force the capabilities of the virtual machine to correspond to the real capabilities of the underlying hardware, and would be errant in doing so. For example, there is software available for SPARC workstations that simulates an Intel processor running the MS-DOS operating system. Even though the hardware cannot directly execute MS-DOS programs, the virtual system containing a SPARC machine and the simulation software could advertise the ability to run MS-DOS programs.

In some cases, there is a one-to-one mapping from nodes to machines, but not always. For example, there is a machine leonardo at Purdue, but there is no machine named CS or Renaissance. These virtual systems can either map onto their own machines, or they can map onto other virtual machines within the distributed system. A machine within a virtual system acts as a representative for that system to higher

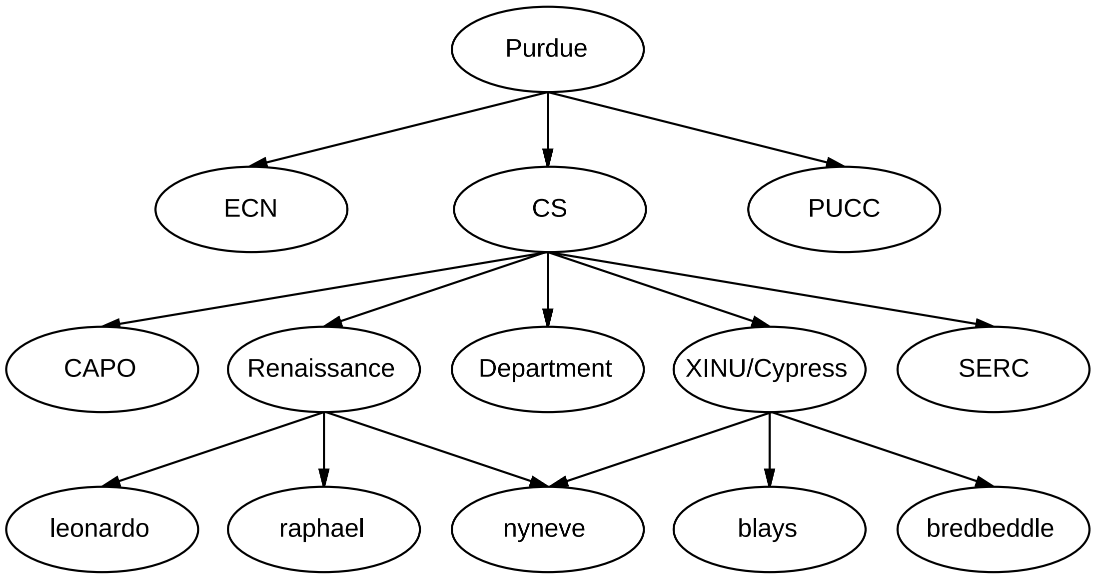

Figure 3.2 The directed acyclic graph for figure 3.1

levels within the hierarchy. Thus, any of nyneve, blays, or bredbeddle might act as the XINU/Cypress spokesman within the CS system.

As defined earlier, virtual systems are hierarchical constructs, where a virtual system is made up of one or more subordinate systems. An encapsulating virtual system is a *parent*, and a subordinate system is a *child*. In the example, CS is the parent of SERC, Renaissance, etc., and they are its children. As is demonstrated by nyneve, a child may have multiple parents. Children with the same parent are called *siblings*. This usage corresponds to the definitions of *son*, *father*, *brother*, *proper ancestor*, and *proper descendant* from [AHU74]. The term *neighbor* refers to one of a node's parents, children, or siblings.

Each virtual system in the hierarchy has a software module (a scheduling support module) that is responsible for maintaining the set of information required by the global scheduling policy and distributing information describing the system state to its neighbors within the graph.[1] This module also controls task execution and

[1] We sometimes use the notation X as a shorthand for "the scheduling module for virtual system X."

movement through the system, and is responsible for collecting data describing the local system. The module provides the mechanism upon which the scheduling policy is built.

Prescribing the system advertisement pattern within the distributed system forces a partial sacrifice of communication autonomy by the participating hosts. Chapter 4 formally defines the rules for inter-node communication of system state information within distributed systems. This prescription is necessary to form a common base for the implementation of scheduling policies.

There are two facets to the local policy that the modules support: task placement and task acceptance. The task placement policy takes a set of tasks and a description of the underlying multicomputer and devises an assignment of tasks to processors according to an optimizing criterion. The task acceptance portion of the policy supports administrative autonomy and execution autonomy by allowing each node to determine its own local acceptance and execution policy.

A state advertisement and request mechanism lies at the heart of the scheduling module. A machine advertises its state through a *system description vector* (SDV) that describes the capabilities and state of a system. When a task is to be scheduled, modules exchange *task description vectors* (TDVs) describing the resource requirements of a task.

3.1.2 The System Description Vector

The system description vector encapsulates the state of a system. A scheduling module uses an SDV to advertise its abilities to other systems that may request it to schedule tasks. Scheduling modules use SDVs as the basis for choosing a candidate system for a task from among their neighboring systems. The system description vector is designed to support the scheduling of conventional tasks, but a flexible extension mechanism permits the tailoring of the vector to other applications.

A review of the scheduling algorithms in section 2.4 yielded a small basis set containing the data most used by the algorithms. Few of these scheduling algorithms use any information beyond processor speed. Of the algorithms in the survey, most use the processor speed as input to their algorithms, while a small number considered the communications structure of the system.

The description vector contains a fixed portion and a variable portion. The fixed portion contains data items supporting the scheduling algorithms from the literature. The variable portion allows administrators to customize the information set in support of specialized policies. The fixed data set includes the following items:

- memory statistics (available and total)
- processor load (queue length, average wait time, and processor utilization ρ)
- processor characteristics (processor speeds, the number of processors)
- a measure of the system's willingness to take on new tasks

The modules automatically determine inter-node communication costs.

This design defines a static set of machine classes for each characteristic. A system that provides special services, such as specialized I/O devices or vector processors can use the extension mechanism described in section 3.2.2.6.

3.1.3 The Task Description Vector

The task description vector is similar to the system description vector—it represents the resource requirements of a task. The task vector is used in conjunction with a system description to decide if a task will be accepted for execution. The task acceptance function can be thought of as a *task filter* that compares the two vectors, subject to the local policy, and decides if a task should be accepted.

The surveyed scheduling algorithms demand specific information about tasks, in contrast to their simplistic demands for system description information. More than half of the 47 algorithms computed results based on the estimated run-time of a task, and more than half used inter-task communication estimates. Therefore, the task description vector consists of the following data items:

- memory requirements
- estimated run time
- originating system
- estimated communications load

Tasks that require special services describe those services using the same extension mechanism used for the system description vectors. Special services might include hardware requirements (vectors processors, specific architectures), operating system requirements (UNIX, VMS), or software requirements (text processors, compilers).

Table 3.1 displays the relationship between fields in the system description vector and corresponding fields in the task description vector. The third column in the table indicates the connection between the fields with expected use by the policy module.

Table 3.1 Comparison of system descriptions to task description

System Description	Task Description	Use by Policy Module
available memory	memory requirements	compare capacity
processor speed		estimate if task will
processor load	estimated time	complete in acceptable time
communication cost	communication load	compare capacity
willingness		used to decide which neighbor to request scheduling from
	originating system	bookkeeping and policy decisions

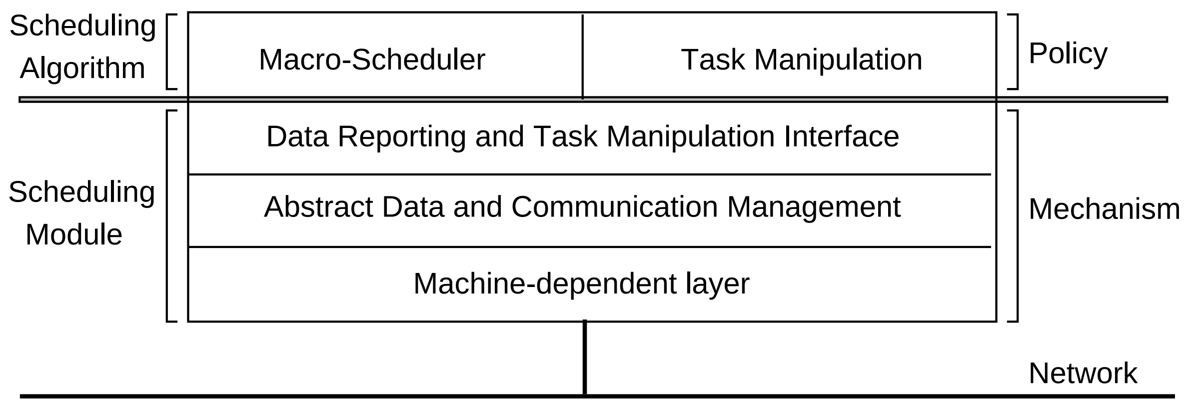

Figure 3.3 The structure of scheduling module

3.2 Module Structure

The structure of a scheduling module is in figure 3.3. Three layers make up the module: the interface layer, the abstract management layer, and the machine-dependent layer. The machine-dependent layer implements communications protocols, task manipulation primitives and data acquisition routines over the native operating system. The abstract management layer uses the machine-dependent layer to communicate with other modules, and provides abstract, architecture and operating system independent operations for data communication and interpretation. The interface layer presents the algorithm implementer with access to the abstract operations in the management layer. Two sample interface layers, a library of function calls and an interpreted language, appear in chapter 5.

3.2.1 The Machine-dependent Layer

The machine-dependent layer fulfills four functions: information encoding, access to network and transport protocols, data acquisition, and task manipulation. The lowest layer abstracts these machine-dependent features and presents them to higher layers.

Information encoding uses an external data representation, as described in section 2.1.1. This layer provides routines to convert the machine-dependent encodings of basic data types into standard format. The data acquisition routines use operating system-specific calls to obtain system state information to fill the SDV.

The communication protocols defined in section 3.2.2.2 assume certain characteristics for network communication. The lowest layer provides access to an datagram-oriented service for the advertisement of system state information, and a reliable protocol for task and data transfer. For the datagram service, an unreliable protocol such as the User Datagram Protocol (UDP) [Pos80a] is acceptable. Examples of reliable protocols include TCP [Pos80b] or a member of the File Transport Protocol (FTP) family [PR85, Sol92, Lot84]. The ISIS system implements levels of service ranging from unreliable messaging protocols at the lowest level to reliable multicast protocols [BJ87].

The choice of protocol depends on the critical characteristic of the channel. For the update channel, timeliness is critical, and reliable protocols typically have higher overhead and delay than unreliable protocols. For the task channel, a reliable protocol ensures delivery of the task and associated data. If an efficient implementation of a reliable messaging protocol exists, such as in later versions of ISIS, then it could be used for the update channel. Section 3.2.2.2 discusses the requirements of the channels in more detail.

In terms of the OSI seven-layer model [DZ83], the machine-dependent layer of the scheduling module contains parts of the session and presentation layers, and provides access to the network and transport layers below. Functions of the OSI application layer appear in the higher layers of the schedule module.

The set of task manipulation primitives contains six members: start, kill, suspend, resume, checkpoint, and migrate. Start begins execution of a program image as a task. Kill aborts a running task. Suspend temporarily stops a running task, and resume restarts a suspended task. Checkpoint saves a task to a program image, and migrate moves a program image between machines.

3.2.2 Abstract Data and Protocol Management

The middle layer provides a uniform implementation of primitives for inter-module communication. It consists of a set of event-based semantics that define inter-module interaction, the communication protocols used by the modules, and the extension mechanism for the description vectors.

3.2.2.1 Event-based Semantics

The support mechanisms use event-based semantics. Figure 3.4 depicts the hierarchy of events. There are three types of events: *finished events*, *timeout events*, and *message events*. Each event has an associated *handler*, which performs actions in response to an occurrence of the event.

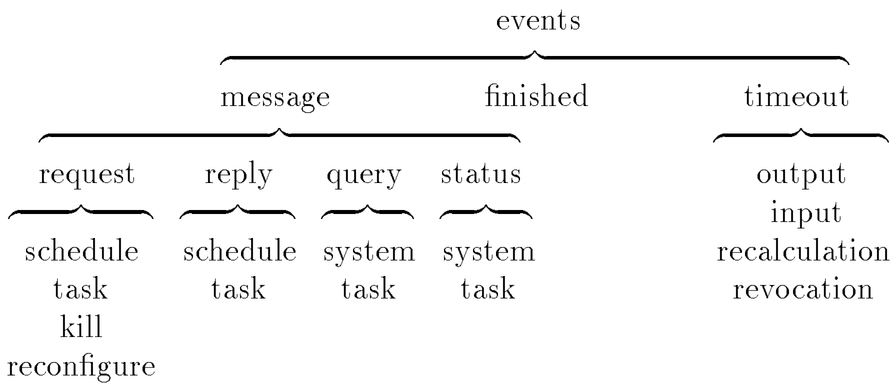

Figure 3.4 Hierarchy of events

A finished event occurs when a task completes execution on the local host. The finished event handler notifies the originating system that the task has completed, and returns any results.

Timeout events occur when a time limit expires. There are four types of timeout events: *output timeouts*, *input timeouts*, *recalculation timeouts*, and *revocation* *timeouts*. When an output timeout occurs, the handler sends a system state advertisement (an SDV) to a neighbor. An input timeout indicates that a neighbor has not advertised its state within the bounds of the period. In response to an input timeout for a neighbor, the module may send a status query to that neighbor. Upon a recalculation timeout, the handler recomputes the update vectors it passes to neighbors. A revocation timeout causes its handler to examine the current host state to see if a task should be revoked.

Message events occur when a message arrives for a module. There are four classes of message events: *request message events*, *reply message events*, *query message events*, and *status message events*. Each of these message event types has subtypes. Request message events ask the handler to perform a service, and comprise *schedule request message events*, *task request message events*, *kill request message events*, and *reconfiguration request message events*. Reply messages occur in response to request messages, and reply message events have two subtypes: *schedule reply message events* and *task reply message events*. Reply message events are paired with the corresponding request message event subtypes.

Query message events and status message events have two subtypes, *task* and *system*. Query events ask the handler to provide for information about tasks and systems, rather than for services to be performed. Status messages contain information describing tasks and systems, and status message events may occur without any query taking place. A complete description of the messages that correspond to message events appears in sections 3.2.2.3 and 3.2.2.4.

A planned extension to the event mechanism includes access points for external agents to trigger events. In this way, the operating system can notify the scheduling module that conditions have changed. For example, the memory manager for the operating system could notify the module that the supply of free memory frames has been exhausted, and trigger a revocation event. Exploitation of this feature would require additional support from the operating system.

3.2.2.2 The Protocols

The communication protocols define the interaction between scheduling modules within the distributed system. All information passing and inter-module coordination takes place through the protocols.

Conceptually, the protocol has three channels: the control, update, and task channels. The update channel advertises system state. The task channel moves a task between systems, and the control channel is used to pass control messages and out-of-band data. The update and control channels only connect neighbors within the distributed system; the task channel may connect any two virtual systems. This sacrifice of communication autonomy is necessary to maintain administrative and communication autonomy at higher levels in the hierarchy.

3.2.2.3 The Update Channel

The update protocol is message-based. Each message contains the system description vector for the sending system, and consists of a message header and a fixed set of data, followed by an optional set of policy-defined data. The interpretation of the policy-defined data is done by the two modules at opposite ends of the channel. The update channel is unidirectional; the recipient of an update message returns no information through the update channel. The update channel makes no attempt to ensure reliability. If a reliable message passing mechanism exists, it may be used. As noted by Boggs, et al. in [BMK88], networks are generally reliable under normal use. Timely delivery of data is more important than reliable delivery; late information is likely to be out-of-date, and therefore of little value. Reliable protocols generally have higher communication overhead than unreliable protocols. Unreliable protocols can also deliver duplicated or out-of-order messages, which can be detected using sequence numbers within the update messages.

The advertisement mechanism operates in one of two modes: polled and timeout-driven. Under polled mode, a system can query another as to its status through the control channel and receive a reply through the update channel.

With timeout-driven updates, the administrator sets the timer for output timeouts. When the countdown timer expires, the scheduling support module advertises the state representation for its virtual system through the update channel. This is done regardless of how recently the module received updates from other systems. The length of the period is a locally tunable parameter, allowing the administrator to determine the tradeoff between overhead and data soundness. A short timeout period ensures that update recipients have an accurate view of the sender, but incurs a penalty of increased machine load. A long timeout is computationally inexpensive, but risks the development of suboptimal schedules based on outdated information.

Update cycles cannot be allowed in the communications structure of a system. An update cycle occurs when an update vector that describes a system is incorporated into another system's update vector and subsequently advertised back to the original system. Such behavior causes an ever-increasing overestimation of system resources, analogous to the *count to infinity* problem in network routing protocols (see Comer [Com91, chapter 15]). For any system, there are three sets of systems that could pass it updates: its children, its parents, and its siblings within the hierarchy. Methods of avoiding overestimation of system resources are discussed in chapter 4.

3.2.2.4 The Control Channel

The control channel is intended to be a bidirectional, reliable, message-based channel, such as the Simple Reliable Message Protocol [Ost93] or the Reliable Datagram Protocol [PH90, VHS84]. A control message consists of a header, including an ID number for the message and a message type, and data that depends on the type of the message. The following defined control message types correspond to message events: *request messages*, *reply messages*, *query messages*, and *status messages*. Each of these message types has subtypes, detailed below.

request messages

schedule

The sending system requests another system to accept a task for execution. This request includes a task description vector for the referenced task.

task

The system requests a task from another system. This request includes a copy of a task description vector describing a task the requester will accept. Receiver-initiated load balancing schemes could use this type of message.

kill

The sender requests that the receiver stop executing the task named in the message. If the receiver chooses to honor the request, it can returns a task_status message with a killed subtype (see below). The receiver is neither obligated to kill the task (execution autonomy), nor to inform the requester if the request was honored (communication autonomy).

reconfigure

The sender requests that the receiver recompute its connectivity, in the event of link failure or dynamic system reorganization. Chapter 4 discusses methods of recomputing connectivity for systems with different DAG-based structures.

reply messages

schedule

The recipient of a schedule request sends a schedule reply having one of two subtypes: accept or deny. An accept subtype indicates that the task has been accepted for execution, and includes the identification number of the accepted schedule request message. The deny subtype indicates that the neighboring system declines to execute the task.

task

The sending system replies to a task request message with a task reply message. Like the schedule reply message, a task reply can have an accept or a deny subtype. An accept subtype means that the sender has a task eligible for migration that matches the description in the original task request message, and the task is moved through the task channel.

A deny subtype indicates that the sender declines to supply a task. The requested system will not migrate a task to the requester; either it is unwilling, or it has no matching tasks. The data includes the identification number of the rejected task request message.

query messages

task status

The sender is requesting information on the status of a task, typically one that it submitted at some point in the past.

system status

The sender is querying the state of a neighboring system. A system description vector may be returned through the update channel in response to this request.

status messages

task status

The sender is responding to a task status query message, or notifying the receiver of the completion of a job. This message can report one of seven possible states: executing, finished, aborted, killed, revoked, denied, and error.

An executing status indicates that the task is still eligible for execution, although it may be blocked. The finished state is sent upon completion of a task, while aborted indicates that the task terminated abnormally, e.g. a bus error or division by zero. The finished state does not guarantee that the task correctly performed its intended function, only that it completed execution.

The killed status indicates that the task was killed on request from the originating system. If the administrator or local policy revokes a task, the scheduling module may send a revoked message. A denied reply means that the requested system would not accept the task for execution.

The error status is returned in the case of a malformed request. Errors in requests include, but are not limited to, querying the status of a task that was scheduled by another system and asking about a nonexistent task. This reply code is intentionally vague, to support future security enhancements by forestalling the acquisition of unauthorized information.

3.2.2.5 The Task Channel

The task channel reliably transfers a task between two nodes in a distributed system. After negotiating a task's destination through the control channel, the module opens a task channel to move the task. This may either be directly between the source and destination, or by a special form of delivery called *proxy transfer*. Proxy transfer is used when the destination is inside a virtual system that prohibits outside systems from directly accessing its members. In this case, the task is delivered to the encapsulating virtual system, which is then responsible for forwarding the task to its destination. Garfinkel and Spafford [GS91] define this type of behavior as a *firewall*. Cheswick discusses the the construction of a secure packet router embodying the firewall concept in [Che90].

3.2.2.6 The Extension Mechanism

It is impossible to predefine the complete set of characteristics used by all present and future scheduling algorithms. Therefore, the description vectors include an extension mechanism that allows users to customize the description of a system or task. Users may append a set of simple values to the description vector, in the form of `(type, variable, value)` triples. The extension mechanism is guaranteed to implement four basic variable types: integers, booleans, floating point numbers, and strings. Aggregate types such as records, arrays, or unions may be available but are not guaranteed to be present.

The extension mechanism is similar in concept to the attribute-based descriptions of the Profile [Pet88] and Univers [BPY90] systems developed at the University of Arizona. We have simplified and tailored the concept to our more limited purposes.

The mechanism is the same for extending both task and system description vectors, but the meaning of the fields is different. The fields describe the requirements of a task or the capabilities of a system. For example, an SDV might contain a boolean variable `latex` with value `true` to indicate that the system has the LaTeX text processing package available, and LaTeX text-formatting tasks could have a boolean variable `latex` set to `true`. The first indicates an offer of service, and the second indicates a request for service. It is up to the system providing the service to correctly interpret the TDV.

3.2.3 The Interface Layer

The interface layer is the implementation vehicle for the scheduling policy, and allows administrators to express policy in terms of the operations provided by the management and machine-dependent layers. The administrator can specify multiple scheduling policies for a single system, and use an adaptive umbrella policy to switch between them based on past behavior of the system or the characteristics of the task.

The architecture does not define the form of the interface layer; any interface that provides access to the internal mechanisms of the module is sufficient. Different modules within the same system can use different interface layers—this is a form of design autonomy.

Two interface layers are described in chapter 5. The first is a library of function calls. An administrator can write his scheduling policy in a high-level language and then compile and link his policy into the scheduling module.

The second layer is an interpreter for a simple language that allows the administrator to construct *filters* to control system behavior. Filters are logical predicates that take as input two description vectors, and return a true or false value based on the contents of the description vector. If a true value is returned, an associated action is taken, such as accepting the task for execution, or selecting a neighbor as a candidate for task scheduling.

3.2.4 Expected Use

The mechanisms are designed for primary support of medium to coarse-grained tasks. The granularity of a task, as defined in [Sto93], is the ratio of a task's computation time (R) to its communication time (C). Coarse-grained tasks have a high value of R/C, and include CPU-bound programs such as traditional scientific computations. Medium-grained tasks include text processing and program compilation. Chapter 6 lists experimental results that validate these assumptions. The mechanisms will support scheduling for fine-grained tasks (those with a small R/C ratio), but will not do so with the same effectiveness as for tasks with higher granularity.

The expected system structure has a small branching factor (< 10) at all levels except at the level immediately above the leaves. The branching factor at the leaves is expected to accommodate a moderate computing cluster, with perhaps a few hundred systems. Thus, the expected use of the mechanisms might include several thousand machines. As will be observed in the next chapter, there are tradeoffs when choosing a system structure. Systems with large branching factors and small depth will have more up-to-date data, but will expend greater resources communicating with other systems. Systems with small branching factors and large depth will have greater delay in message propagation between remote parts of the system, but the communication load on each individual system will be lessened.

3.3 Execution of a Scheduling Module

As a simple example of how the individual layers interact, we will describe the hypothetical execution of a scheduling module implementing a simple policy. At an initial steady state, there are no tasks running on the local system, and the scheduling module has received update messages from its neighbors, and therefore has a *view* of each neighbor, consisting of the SDV advertised by the neighbor.

A user on the local host submits a task to the distributed system, causing a schedule request event. Included in the request is a TDV describing the task. The scheduling module invokes a task filter, comparing the TDV with the SDV of each neighbor as well as the SDV for itself, and determines which SDV most closely matches the TDV according to the local policy. If the local system's SDV most closely matches, the local system accepts the task. If the SDV most closely matches a neighbor, the request is forwarded to the neighbor. If the schedule request had come from a neighbor rather than a user on the local system, the module would have behaved similarly, except that the neighbor that submitted the request would not be considered by the task filter.

Assume that the local module accepts the task. The scheduling module opens a task channel to the client program, and transfers the task and associated data to the local host. The task begins executing, and the module enters bookkeeping data regarding the task.

After a short time, a recalculation timeout event occurs, the scheduling module uses the data-collection functions to determine the local system state, and stores it in an SDV. The module then creates an update message to send to each neighbor, based on its own SDV and the SDVs it has received from its neighbors. The exact contents of the SDV contained within the update message can depend on the destination of the message; chapter 4 discusses the combining rules used to form update messages.

A revocation timeout occurs, and the module checks the system state against the local policy, and determines that no tasks need to be revoked. If the policy had dictated that tasks needed to be revoked, perhaps because of increased system load caused by interactive users, a revocation filter would have determined likely candidates for revocation.

A few moments later, an output timeout occurs, and the module sends the actual update messages computed during the handling of the recalculation event. Shortly thereafter, the task completes execution, and the module returns the results to the client program, and returns to the steady state.

3.4 Summary

This chapter has outlined an architectural model that may be used to construct distributed, autonomous, heterogeneous systems. It explained the role of scheduling modules within the larger system, and described the operations and inter-module communication protocols for these modules.

4. FORMAL MODELS FOR SYSTEM STATE DISSEMINATION

Previous chapters describe the description vector dissemination mechanism and introduce modeling of distributed systems with directed acyclic graphs. This chapter gives three sets of rules that govern the flow of update information through the system, and proves that the rules are sound.

If schedule writers are to have confidence that their schedulers function correctly, the underlying mechanisms must perform properly. In the case of information dissemination, we have chosen proper performance to mean that the support mechanisms guarantee that information advertised by a host, through its update channel, reaches every other host within the system exactly once.

There are two aspects of exactly once semantics: guaranteeing that advertised information reaches all other nodes within the system, defined here as *completeness*, and guaranteeing that information is not duplicated during advertisement, defined here as *correctness*. The correctness constraint arises from one of the basic assumptions for the information dissemination mechanisms: overestimation of system resources must be avoided.

These guarantees apply only to the underlying mechanism; because of communication autonomy, nodes within the distributed system may choose not to forward information they have received, thus preventing the advertisement from reaching some nodes within the system. Nodes could also advertise false information, thus overestimating resources. This is acceptable because it is a policy decision; the proofs in this chapter guarantee only that the underlying mechanism is sound.

We define two subtypes of completeness: *global completeness* and *local completeness*. Information advertised with a globally complete state dissemination mechanism reaches all other nodes within the entire distributed system, while a locally complete

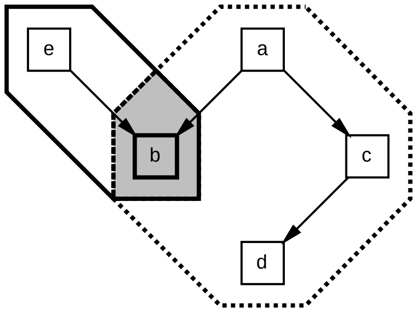

Figure 4.1 Global and local completeness

mechanism guarantees that advertised information reaches all other nodes with the same root.

As an example distinguishing local completeness from global completeness, consider figure 4.1. The system rooted at a contains the nodes $\{a, b, c, d\}$, and the system rooted at e contains the nodes $\{b, e\}$. With a globally complete mechanism, information advertised by e will reach a, b, c, and d. With a locally complete mechanism, it will only reach b. Similarly, with a globally complete mechanism, an advertisement by d reaches e, but not with a locally complete mechanism.

At first glance, a locally complete mechanism might seem less desirable than a globally complete mechanism. However, the administrative hierarchies that give rise to the system structure provide a motivation for local completeness. In terms of administrative domains, b is a shared resource jointly administered by e and a. By joining with e to administer b, a has not granted e the right to use a, c, or d. Local completeness allows administrators to automatically restrict data advertisements and scheduling requests to a rooted subgraph, provides autonomy support, and can form the basis for security mechanisms.

This chapter defines operations for combining update vectors and analyzes the semantics of the operation. Section 4.2 defines notation that is common to all the combining rules. Section 4.3 proves that for tree-structured distributed systems, the update mechanism is correct and globally complete. Section 4.4 refines the update mechanism, and proves the refined mechanism correct and locally complete for a class of non-tree-structured systems. Section 4.5 defines a globally complete and correct mechanism for systems structured as general DAGs.

4.1 Assumptions About Policy

This chapter proves that the update mechanism is sound, and proves that if accurate and complete information is supplied to the mechanisms, then accurate and complete information will be disseminated throughout the distributed system. However, sound mechanism does not preclude the advertisement of erroneous system descriptions, whether by accident or malice.

The model for update flow is that a module collects several description vectors, adds information describing the local system, and condenses the resulting set of description vectors into one vector. During this process, the system can corrupt the information content of the vector in one of three ways: the system can underestimate resources, overestimate resources, or change values within the description vectors that do not represent quantities.

Underestimation of resources can hinder completeness, but is necessary to support communication autonomy. For example, a research group might wish to make a group of workstations available to outside agencies for general purpose computation, but restrict access to a parallel processor to members of the research group. The support for communication autonomy within the mechanisms allows the research group to restrict advertisement of the parallel processor's resources to other virtual machines under the administrative control of the research group. In any case, underestimation of resources will not cause tasks to be erroneously scheduled, but it might leave available resources unused.

Overestimation of resources can cause task requests to be misdirected, and can result in inefficient schedules and execution delays. However, because the mechanisms do not include any sort of voting procedure, the mechanisms have no way of verifying that a virtual system actually possesses the resources it advertises, and must accept and propagate the advertisement.

Changing of data, such as altering strings containing file names, can cause results similar to overestimation. Omitting non-quantitative data can cause effects similar to underestimation.

Therefore, the communication mechanisms assume that systems can and will underestimate resources in support of communication autonomy. However, systems are not expected to overestimate or change resource information. Policies can be written that violate these assumptions, but no assurances can be made about the performance of such policies.

4.2 Notation

This chapter uses a multiset notation to denote the system description vectors passed between systems (the update vectors from section 3.1). A multiset is a collection of similar elements, and may contain duplicate elements. Even though the actual data passed through the update vector is untagged, the proofs use the name of a system to represent its capabilities in the multiset. For example, the set $\{a, b, c\}$ represents a description vector that contains the capabilities of the distributed system combining a, b, and c. The set $\{Renaissance, leonardo, raphael, nyneve\}$ represents the capabilities of the Renaissance virtual system from figure 3.2. This notation makes it obvious when a description vector violates the correctness constraint by including a node's capabilities multiple times, because the name of the node appears multiple times in the multiset.

We define four operators on multisets: $\uplus$, $\cap$, $\setminus$, and $\|\,\|$. The operator $\uplus$ is the multiset union operator with duplicate inclusion. For example, $\{a, b\} \uplus \{a, c\} = \{a, a, b, c\}$. The $\uplus$ operator coalesces the representation of two system description vectors into one. The notation $\biguplus_{i \,\in\, range} \mathcal{S}_i$ represents the mapping of the $\uplus$ operator over multiple multisets, and ϕ denotes the empty multiset. $\setminus$ is the difference operator for multisets, and is defined to remove as many instances of an item from the first set as appear in the second set, e.g. $\{a, a, b, c\} \setminus \{a, b, d\} = \{a, c\}$. The intersection operator for multisets, $\cap$, yields a multiset containing the lesser number of each element common to two multisets, e.g. $\{a, b\} \cap \{b, b, c\} = \{b\}$. The $\|\|$ operator returns the number of elements in the set, e.g. $\|\{a, b, c\}\| = 3$.

4.3 Tree-structured Systems

This section defines and proves properties of update semantics for tree-structured systems. We use standard definitions from graph notation, as found in [AHU74]. A tree is a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with the following three properties:

1. There is exactly one vertex, called the root, that has no parent.
2. Every vertex except the root has exactly one parent.
3. There is a path from the root to each vertex; furthermore, this path is unique.

An individual edge in the graph is denoted by listing its endpoints, e.g. an edge from vertex v_1 to vertex v_2 is written (v_1, v_2).

The tree-structure of these graphs is motivated by existing administrative domains. The typical administrative domain within an organization is tree-structured, and these rules are optimized for the expected case.

4.3.1 Combining Rules

For a node x, equations 4.1 through 4.5 give definitions for the sets containing the parent of x (Pa_x), the children of x, (Ch_x), the siblings of x (Si_x), the ancestors (An_x), and the descendants (De_x) of x. Note that with tree-structured systems, $\|Pa_x\|$ equals 1.

$$Pa_x = \{p \mid (p, x) \in \mathcal{E}\} \tag{4.1}$$

$$Ch_x = \{c \mid (x, c) \in \mathcal{E}\} \tag{4.2}$$

$$Si_x = \{s \mid (\exists p \mid (p, x) \in \mathcal{E}, (p, s) \in \mathcal{E}, s \neq x)\} \tag{4.3}$$

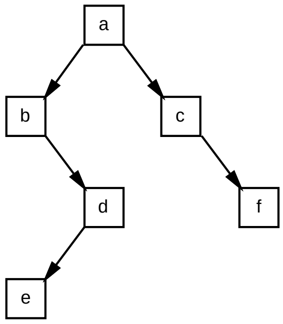

Figure 4.2 A tree-structured distributed system

$$An_x = \{a \mid a \in Pa_x \text{ or } (\exists p \mid p \in Pa_x, a \in An_p)\} \tag{4.4}$$

$$De_x = \{d \mid x \in An_d\} \tag{4.5}$$

Each system computes two sets of system status data to be passed to its neighbors. The U set is passed *up* to the system's parents and sideways to its siblings, and the D set is passed *down* to its children. The notation U_x and D_x refers to the sets maintained by a system x within a tree-structured system. Equations 4.6 and 4.7 define these recursively in terms of the structure of the system. The total data stored for outgoing update vectors is independent of the number of children or siblings for a node, which helps to bound the resource usage of the mechanisms.

$$U_x = \{x\} \uplus \left(\biguplus_{i \in Ch_x} U_i \right) \tag{4.6}$$

$$D_x = \{x\} \uplus \left(\biguplus_{j \in Pa_x} D_j \right) \uplus \left(\biguplus_{k \in Si_x} U_k \right) \tag{4.7}$$

Informally, the U vectors include the description of the node and the U vectors from all its children. The D vector includes a node and the D vector from its parent and the U vectors from all of its siblings. The U and D vectors for the example system in figure 4.2 are in table 4.1.

Table 4.1 Example U and D sets

$U_a = \{a, b, c, d, e, f\}$	$D_a = \{a\}$
$U_b = \{b, d, e\}$	$D_b = \{a, b, c, f\}$
$U_c = \{c, f\}$	$D_c = \{a, b, c, d, e\}$
$U_d = \{d, e\}$	$D_d = \{a, b, c, d, f\}$
$U_e = \{e\}$	$D_e = \{a, b, c, d, e, f\}$
$U_f = \{f\}$	$D_f = \{a, b, c, d, e, f\}$

4.3.2 Proofs of Semantics

This section contains proofs that the mechanisms defined in equations 4.6 and 4.7 are globally complete and correct.

To prove that the semantics are complete, we must prove that a datum advertised by a node will reach all other nodes in the tree. To prove that the semantics are correct, we must prove that datum reaches other nodes at most once.

The following steps provide an outline of the proof: 1) prove that a system appears in a node's U vector if and only if the system is a member of the subtree rooted at the node, 2) prove that a system appears in a node's D vector if and only if that system is not a descendant of the node, 3) prove that no system appears more than once in a U vector, and 4) prove that no system appears more than once in a D vector. Steps one and two prove completeness, and steps three and four prove correctness of the semantics for tree-structured systems.

The first lemma states that the U vector for a node represents the node and its descendants, and only the node and its descendants.

LEMMA 4.1 $(y \in U_x) \Leftrightarrow (y = x \text{ or } y \in De_x)$
(a system appears in a node's U vector if and only if the system is a member of the subtree rooted at the node.)

> Proof (by induction): Recall the definition of U_x in equation 4.6. First comes the proof of the $\Rightarrow$ implication.

For a leaf system, $U_x = \{x\}$, and the implication holds. This is the base case. For a non-leaf system, the induction step assumes that the implication is true for all children of x. Thus, $\forall c \in Ch_x$, U_c contains only descendants of c, and c itself. The second term of equation 4.6 adds only children of x and their descendants, which are also descendants of x. Therefore the second clause of the implication holds, and the $\Rightarrow$ case is true.

Now for the $\Leftarrow$ implication:

By definition, x is always in U_x, so we need only prove that all descendants of x are members of U_x. At a leaf node there are no descendants, so the base case is proven. For the induction step, we assume that the implication holds for all children of x. Therefore, the second term in the definition of U_x adds all children of x, and all of their descendants, which is to say it adds all the descendants of x. Thus, $y \in De_x \Rightarrow y \in U_x$. $\square$

The next lemma states that any system that appears in D_x is not a descendant of x, and also that all non-descendants of x appear in D_x. This lemma will be used to establish the completeness property for tree-structured graphs.

LEMMA 4.2 $y \in D_x \Leftrightarrow y \notin De_x$

First comes the proof of the $\Rightarrow$ implication.

Proof (by induction): At the root, $D_x = \{x\}$. By definition, $x \notin De_x$. For the induction step, assume that the lemma is true for the parent of the node x. Suppose that the lemma is false for x, i.e. $\exists y$ such that $y \in D_x$ and $y \in De_x$, and examine each of the three terms in the definition of D_x. y cannot be x, as $x \notin De_x$ by definition. y cannot come from the second term, as that would violate the induction assumption. From lemma 4.1 and the fact that the U vectors for siblings are disjoint because of the tree-structure of the graph, y cannot come from the third term. Therefore, y cannot exist, and the lemma is true for x in the inductive step. □

Now comes the proof of the $\Leftarrow$ implication.

Proof (by induction): At the root, $D_x = \{x\}$. Because x is the root, all other nodes are descendants of x, so the base case is true. For the induction step, we partition the set of non-descendants of x into two groups: those that are not descendants of x's parent, p, and those that are descendants of p. The induction step assumes the implication holds for p.

All non-descendants of p are also non-descendants of x, and because of the induction assumption, are included in D_x by the second term in the definition of D_x. This leaves the descendants of p for consideration. From the $\Leftarrow$ implication of lemma 4.1, all siblings of x and their descendants are included by the third term in D_x. By definition, $x \in D_x$. Therefore, all descendants of p that are not descendants of x appear in D_x.

Therefore, all non-descendants of x appear in D_x. □

THEOREM 4.1 The Completeness Theorem for Tree-Structured systems: Information describing each node reaches every other node within the system, using rules 4.6 and 4.7.

Proof: By lemma 4.2, information describing all non-descendants of a node x appears at x. By lemma 4.1, information describing all descendants of x is visible to x. As x knows about itself, x receives a description of all nodes in the system, and the semantics are (globally and locally) complete. □

The next lemma is crucial in establishing correctness. This lemma states that U vectors do not contain duplicate entries.

LEMMA 4.3 $\forall y \in U_x, y \notin (U_x \setminus \{y\})$
(No system is represented more than once in a U vector.)

> Proof (by induction): At a leaf, $U_x = \{x\}$, and the lemma is true. Assume that the lemma is true for all children of a node x.
>
> If the lemma is false, then $\exists y \mid y \in (U_x \setminus \{y\})$. Based on equation 4.6, either $y = x$ and $x \in U_c$ for some child c of x, y appears in the U vectors of multiple children of x, or a child of x has duplicates in its U vector. From lemma 4.1, $x \in U_c \Rightarrow x \in De_x$, which cannot be in an acyclic graph. Likewise, the tree-structure of the graph means that the U vectors for two children of a node are disjoint, so y cannot appear in the U vectors of multiple children of x. The induction assumption precludes the third possibility. Therefore, y cannot exist, and the lemma is proven. $\square$

The final lemma in this section states that D vectors do not contain duplicate entries. This lemma is also vital to proving correctness.

LEMMA 4.4 $\forall y \in D_x, y \notin (D_x \setminus \{y\})$
(No system is represented more than once in a D vector)

> Proof (by induction): At the root, $D_x = \{x\}$, and the lemma is true. For the induction assumption, assume that the lemma is true for the parent of a node.
>
> If the lemma were false, then $\exists y \mid y \in (D_x \setminus \{y\})$. Based on equation 4.7, one of the following must be true:
>
> 1. $x \in D_p$ for $p \in Pa_x$ (interaction of the first and second terms)
> 2. $x \in U_k$ for some $k \in Si_x$ (interaction of the first and third term)

3. $D_p \cap U_k \neq \phi$ for $p \in Pa_x$ and some $k \in Si_x$ (interaction between the second and third terms)

4. $U_j \cap U_k \neq \phi$ for some $j, k \in Si_x, j \neq k$ (interaction between two members of the third term)

Lemma 4.3 and the induction assumption eliminate the possibility of duplicates within a single incoming vector. By lemma 4.2, (1) cannot be true. For x to appear in a sibling's U vector, x would must be a descendant of its sibling, which violates the tree structure of the graph; this eliminates case (2). For (3) to hold, a system simultaneously be a descendant of p (corollary of lemma 4.1 and the fact that $k \in Ch_p$) and also not be a descendant (lemma 4.2). Therefore, (3) cannot hold. The disjointedness of U vectors for siblings eliminates case (4).

Thus, y cannot exist and the lemma is proven. □

THEOREM 4.2 The Correctness Theorem for Tree-Structured Systems:
No system's attributes appear in any update vector more than once, using rules 4.6 and 4.7.

Proof: Lemma 4.3 states that no system appears more than once in a U vector. By Lemma 4.4, no D vector has duplicate entries. From lemmas 4.1 and 4.2, the D and U vectors arriving at a node are disjoint. Therefore, the semantics for combining update vectors will not overestimate system resources, and are thus correct. □

In theorems 4.1 and 4.2, we have shown that the update protocol presented here will accurately disseminate system description information through a tree-structured distributed system.

4.4 A Subclass of Non-tree-structured DAGs

Equations 4.6 and 4.7 cannot be applied to more general directed acyclic graphs—if a node has two parents with a common ancestor, the capabilities of the system will be overestimated (see the example in figure 4.3). The resulting erroneous U and D sets are in table 4.2. D_d violates the correctness constraint, because it overstates resources; the two paths between a and d cause this.

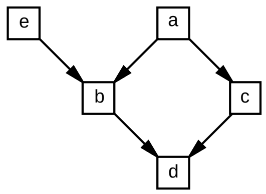

Figure 4.3 A non-tree-structured system

Table 4.2 Erroneous U and D sets for figure 4.3

$U_a = \{a, b, c, d, d\}$	$D_a = \{a\}$
$U_b = \{b, d\}$	$D_b = \{a, b, c, d, e\}$
$U_c = \{c, d\}$	$D_c = \{a, b, c, d\}$
$U_d = \{d\}$	$D_d = \{a, a, b, b, c, c, d, d, d, e\}$
$U_e = \{b, d, e\}$	$D_e = \{e\}$

4.4.1 A Refinement: Primary Parents

Introducing the notion of *primary parents* for each system and modifying the U and D vector definitions solve this problem. A node may have more than one primary parent, but no two primary parents of a node may share a common ancestor. For example, in figure 4.3, b has two parents, a and e, which do not have a common ancestor, so both a and e may be primary parents of b. In contrast, both of d's parents, b and c, have a common ancestor in a, and therefore they cannot both be primary parents for d.

A *primary link* joins a primary parent to a child. A *primary path* is a path composed only of primary links. The proofs and semantics presented here assume that the primary parents have already been selected, and that these links form a spanning tree for the graph. As a result, a primary path exists from each node to all of its roots, and there is exactly one primary path between any two nodes within the system. Note that the spanning tree does not reduce the proofs to the previous case, as the spanning tree is undirected, while the links within the original graph are directed. Section 4.4.3 discusses rules for constructing an appropriate spanning tree.

The notions of primary ancestor, primary descendant, and primary sibling are analogous to the notions of ancestor, descendant, and sibling defined earlier. Equations 4.8 through 4.12 define the sets of primary parents, children, siblings, ancestors, descendants, and roots for a node.

$$PP_x = \{p \mid (p,x) \in \mathcal{E}, \text{ and } (p,x) \text{ is a primary link}\} \tag{4.8}$$

$$PC_x = \{c \mid (x,c) \in \mathcal{E}, \text{ and } (x,c) \text{ is a primary link}\} \tag{4.9}$$

$$PS_x = \{s \mid (\exists p \mid p \in PP_s, p \in PP_x, s \neq x)\} \tag{4.10}$$

$$PA_x = \{a \mid a \in PP_x \text{ or } (\exists p \mid p \in PP_x, a \in PA_p)\} \tag{4.11}$$

$$PD_x = \{d \mid x \in PA_d\} \tag{4.12}$$

$$PR_x = \{r \mid (r = x \text{ or } r \in PA_x), PP_r = \phi\} \tag{4.13}$$

As noted in a previous section, non-tree-structured graphs occur in practice when two systems share administrative control of a third system (e.g. a machine jointly administered by two research projects). The combining rules for primary parents resemble those for tree-structured systems, with the following exceptions: only primary parents and primary siblings of a node incorporate U updates from that node; similarly, only primary children of a node incorporate its D updates. Other parents, children, and siblings receive the updates and can use them to make scheduling decisions, but do not include them in the computation of their own update vectors.

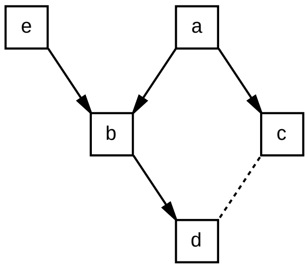

Figure 4.4 A non-tree-structured system with primary parents

Equations 4.14 and 4.15 incorporate the primary parent into the definitions of the U and D vectors. As in tree-structured systems, the data storage requirements for outgoing vectors are constant.

$$U_x = \{x\} \uplus \left(\biguplus_{i \in PC_x} U_i \right) \tag{4.14}$$

$$D_x = \{x\} \uplus \left(\biguplus_{j \in PP_x} D_j \right) \uplus \left(\biguplus_{k \in PS_x} U_k \right) \tag{4.15}$$

In figure 4.4, the solid arrows represent the primary links, and the dashed line is a non-primary link between c and d. The U and D vectors for figure 4.4 are in table 4.3.

Table 4.3 U and D sets for figure 4.4

$U_a = \{a, b, c, d\}$	$D_a = \{a\}$
$U_b = \{b, d\}$	$D_b = \{a, b, c, e\}$
$U_c = \{c\}$	$D_c = \{a, b, c, d\}$
$U_d = \{d\}$	$D_d = \{a, b, c, d, e\}$
$U_e = \{b, d, e\}$	$D_e = \{e\}$

4.4.2 Proofs of Semantics for Primary Parents

This section proves that the semantics defined in equations 4.14 and 4.15 are correct and locally complete for DAG-structured systems that fulfill the constraints outlined previously.

Once again, there are four steps to the proofs of correctness and completeness: 1) prove that a system appears in a node's U vector if and only if the system is a member of the primary subtree rooted at the node, 2) prove that a system appears in a node's D vector if and only if that system is not a primary descendant of the node, but shares a common primary root with the node, 3) prove that no system appears more than once in a U vector, and 4) prove that no system appears more than once in a D vector.

The first lemma states that the U vector for a node represents the node itself, plus all the node's primary descendants. In addition, the lemma states that the U vector represents no other systems.

LEMMA 4.5 $(y \in U_x) \Leftrightarrow (y = x \text{ or } y \in PD_x)$
A system appears in the U vector for a node if and only if the system is a primary descendant of the node, or the system and the node are the same.

> The proof follows the same form as that for lemma 4.1, considering primary descendants instead of descendants.

The next step in proving local completeness is to prove that a host is represented in a node's D vector if and only if the host is not a primary descendant of the node and the host and the node share a common primary root. In the case of tree-structured graphs, the second condition was met by all nodes because there was a single root. In the more general case, there may be multiple roots, so the proof of this lemma does not follow directly from the proof for tree-structured graphs.

LEMMA 4.6 $y \in D_x \Leftrightarrow (y \notin PD_x)$ and $(PR_x \cap PR_y \neq \phi)$

> Proof (by induction):
>
> $\Rightarrow$ implication:
>
>> At the root, $D_x = \{x\}$. By definition, x is not a primary descendant of itself. Also by definition, x is its own primary root, so the lemma holds in the base case.

For the induction step, assume that the lemma is true for all primary parents of the node x. Choose a node $y \in D_x$. If $y = x$, the implication is true from the definition of PD_x. If y comes from the second term in the definition of D_x, then the induction assumption applies. If y comes from the third term, then from lemma 4.5, y is not a primary descendant of x. Because any two nodes that share a primary ancestor share a primary root, $PR_y \cap PR_x \neq \phi$. Therefore, the $\Rightarrow$ implication holds.

$\Leftarrow$ implication:

At the root x, the only node not a primary descendant of x that shares a primary root with x is x itself, and $x \in D_x$. This is the base case.

For the induction step, assume the implication is true for all parents of x. From the assumption, all nodes with the same roots as x that are not descendants of x's primary parents are included in D_x. From the definition of D_x, lemma 4.5, and the spanning tree imposed by the primary links, all primary siblings of x and their primary descendants appear in D_x. Therefore, the $\Leftarrow$ implication is true.$\Box$

THEOREM 4.3 The Local Completeness Theorem for Primary Parents: Information describing each node reaches every other node with the same primary root, under rules 4.14 and 4.15.

Proof: By lemma 4.6, information describing all non-descendants of a node x, that share a primary root with x, appears at x. By lemma 4.5, information describing all descendants of x is visible to x. Because x knows about itself, x receives a description of all nodes in the system with the same primary root. $\Box$

The proof of correctness for primary parents has two subparts, showing that neither U nor D vectors have duplicate entries.

LEMMA 4.7 $\forall y \in U_x, y \notin (U_x \setminus \{y\})$
(No system is represented more than once in a U vector.)

> This proof follows the same form as the proof for lemma 4.2, considering primary children instead of children.

Proving that no system is represented more than once in a D vector is more difficult for primary parents than for tree-structured systems, because of the interplay possible between the D vectors of multiple primary parents.

LEMMA 4.8 (No system is represented more than once in a D vector)
$\forall y \in D_x, y \notin (D_x \setminus \{y\})$

> Assume $\exists y | y \in (D_x \setminus \{y\})$. Then, based on equation 4.15, one of the following must be true:
>
> 1. $x \in D_p$ for some $p \in PP_x$ (interaction of the first and second terms)
> 2. $x \in U_k$ for some $k \in PS_x$ (interaction of the first and third term)
> 3. $D_m \cap U_k \neq \phi$ for $m \in PP_x$ and some $k \in PS_x$ (interaction between the second and third terms)
> 4. $D_j \cap D_k \neq \phi$ for any $j, k \in PP_x, j \neq k$ (interaction between two members of the second term)
> 5. $U_j \cap U_k \neq \phi$ for any $j, k \in PS_x, j \neq k$ (interaction between two members of the third term)
>
> By lemma 4.6, (1) cannot be true. Lemma 4.5 and the imposition of a spanning tree by the primary parent links eliminate cases (2) and (5). For (3) to hold, a system must simultaneously be a descendant of p (corollary

of lemma 4.5 and the fact that $k \in PC_p$) and also not be a descendant (lemma 4.6). Therefore, (3) cannot hold. If (4) were true, then the two primary parents j and k must share a common root (lemma 4.6), and thus could not both be primary parents of x because of the spanning tree imposed by the primary links.

Thus, a contradiction is reached and y cannot exist, so the lemma is proven. □

THEOREM 4.4 The Correctness Theorem for Primary Parents: No system's attributes appear in any update vector more than once using rules 4.14 and 4.15.

Proof: Lemma 4.7 states that no system appears more than once in a U vector. By lemma 4.8, no D vector has duplicate entries. From lemmas 4.5 and 4.6, the vectors arriving at a node are disjoint. Therefore, the semantics for combining update vectors will not overestimate system resources, and are thus correct. □

Theorems 4.3 and 4.4 show that the update protocol presented here will correctly and with local completeness disseminate system description information through a DAG-structured distributed system that meets the constraints detailed in the next section.

4.4.3 Constraints on Primary Parents

The proofs in the previous section assume that a spanning tree has already been imposed on the graph representing the system. However, this may not always be possible to do and still retain local completeness. This section defines a subclass of directed acyclic graphs for which spanning trees can be imposed without loss of local completeness. A spanning tree that preserves local completeness is called a *viable spanning tree*, and a graph with a viable spanning tree is a *viable graph*.

We first examine situations in which the primary parent mechanism fails. For example, consider figure 4.5. The graph in part (a) represents a distributed system with

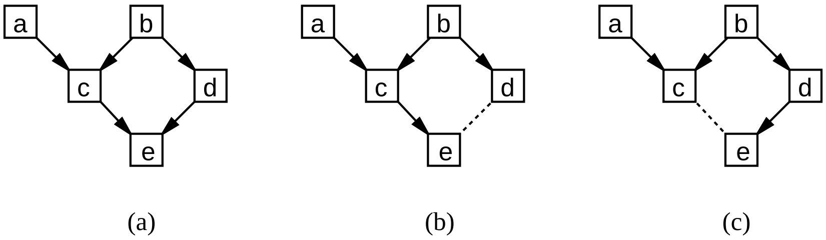

Figure 4.5 A distributed system and two spanning trees

multiple spanning trees. Subfigures (b) and (c) represent two choices. e must choose either c or d as its primary parent. Choosing c as the primary parent, represented in subfigure (b), allows local completeness, because each node in the spanning tree is a primary descendent of all roots of which it was a descendant in the original graph. In subfigure (c), e is not a primary descendant of a, so information describing e will not reach a.

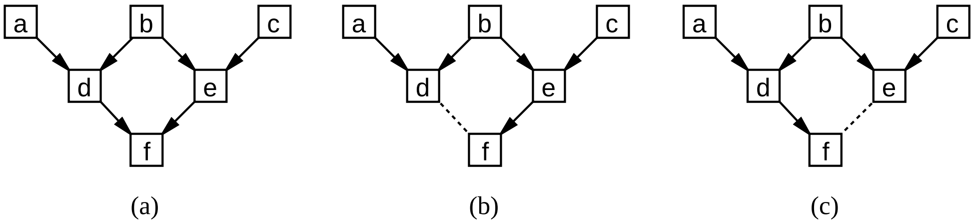

Figure 4.6 A distributed system with no viable primary parents

The graph depicted in figure 4.6 part (a) has no spanning tree that allows for local completeness. If f makes e its primary parent (subfigure (b)), then a will not receive a description of f. If f makes d its primary parent (subfigure (c)), then c will not receive information describing f. There is no viable spanning tree for this graph.

To preserve completeness, it is necessary that a directed path from each of a node's roots to the node still exist in the subgraph covered by the spanning tree. Formally, the following predicate defines the viability property of a directed graph.

$$viable(\mathcal{V},\mathcal{E}) \Leftrightarrow \forall v \in \mathcal{V}, \text{ there exists a subset } \mathcal{P} \text{ of } Pa_v \text{ such that } \left(\bigcup_{p\in\mathcal{P}} Ro_p\right) = Ro_v; \forall p,q \in \mathcal{P} Ro_p \cap Ro_q = \phi; \text{ and the primary links form a spanning tree for the graph.} \tag{4.16}$$

The first part of this definition is a statement of viability in terms of set covering, and determining the first property of viability for a graph is equivalent to determining an exact cover for a node's root set by its parents' root sets. This is known to be an NP-complete problem (see [AHU74]). The second part of the definition ensures that the primary links selected by the set covering form a spanning tree.

The remainder of this section describes and analyzes a method for determining if a graph is viable. First, each node must be able to compute its Ro_x set. This can be accomplished by having each node annotate its D^p vector with a label indicating its Ro_x set. At a root, $Ro_x = \{x\}$. The root set of any other node is the union of its parents' root sets, i.e. $Ro_x = \bigcup_{p\in Pa_x} Ro_p$.

Figure 4.7 contains a straightforward $O(2^n)$ algorithm, where n is the number of parents of a node, to test all possible combinations of the parent sets for coverage. The number of parents for a node is expected to be small, so this exponential growth is acceptable. This algorithm assumes that the root sets of the parents are stored in the set $\mathcal{R} = \{R_1, R_2, \ldots, R_n\}$. The algorithm takes as input the set $\mathcal{R}$, a cover set $\mathcal{C}$, a set of primary parents $\mathcal{P}$, a root set to be covered Ro, and two integers to act as counters and limit variables. It returns the list of parents that create the cover set, if an exact cover is possible. A return value of ϕ indicates failure.

Each node can independently determine if an exact cover set of its root set by its parents' sets exists with a call

$$\text{check\_viable}(\textstyle\bigcup_{p\in Pa_x}\{Ro_p\},\ \phi,\ \phi,\ Ro_x,\ 1,\ \|Pa_x\|).$$

check_viable($\mathcal{R}$, $\mathcal{C}$, $\mathcal{P}$, Ro, i, lim)

1. if ($\mathcal{C} = Ro$) return $\mathcal{P}$;
2. for j in i . . . lim {
3.   if ($\mathcal{C} \cap R_j = \phi$) {
4. $\mathcal{P}' \leftarrow$ check_viable($\mathcal{R}$, $\mathcal{C} \cup R_j$, $\mathcal{P} \cup \{j\}$, Ro, j+1, n);
5.   if ($\mathcal{P}' \neq \phi$) return $\mathcal{P}'$;
6.   }
7. }
8. return ϕ;

Figure 4.7 An algorithm to compute set covering for primary parents

This can be done in parallel, and the parallel running time is $O(2^m)$, where $m = max(\|Pa_v\|), \forall v \in \mathcal{V}$. If the maximum number of parents for a node is fixed, this becomes a constant-time algorithm.

To verify that the primary links associated with the resultant primary parent set form a spanning tree for the graph, it is sufficient to show that there is unique path between nodes.

Duplicate-path detection can be accomplished by broadcasting a status message across the reliable control channel with a unique token across all primary links. Nodes that receive the message broadcast it out over all primary links except the one over which the message was received. If the set of primary links forms a spanning tree, no node will receive the message twice. If a node receives the message twice, it can send an error reply to the originator of the message, indicating that the system configuration is incorrect. Thus, a host can determine if a viable spanning tree will exist when the host attempts to join a distributed system.

4.5 General Directed Acyclic Graphs

Prior sections have described combination rules for certain subclasses of directed acyclic graphs. This section gives globally complete and correct combination rules for general DAGs, with the tradeoff of higher computational and communication overhead.

4.5.1 General Combination Rules

First, a spanning tree is imposed on the graph using primary parents. The viability constraint of the previous section does not apply; any spanning tree will suffice. Unlike the previous semantics, these rules do not pass updates to siblings. Also, instead of sending the same vector to multiple neighbors, these rules require a distinct update vector for each parent or child. Thus, the storage and computational requirements are $O(\|Pa_x\| + \|Ch_x\|)$ for a node x. The rules for the update vector U are in equation 4.17. Note that the name U does not indicate a direction of passage in this case.

$$U_{xn} = \{x\} \uplus \left(\biguplus_{p \,\in\, PP_x, p \neq n} U_{px} \right) \uplus \left(\biguplus_{c \,\in\, PC_x, c \neq n} U_{cx} \right) \tag{4.17}$$

The effect of this rule can be summarized as follows:

> Over a link from x to y, send out information describing x and the sum of information received by x on all other links.

Equation 4.17 employs the technique of a *split horizon update* found in network routing protocols [Hed88, Mal93].

4.5.2 Proofs of General Rules

This section proves rule 4.17 globally complete and correct, given that a spanning tree has been imposed on the graph.

THEOREM 4.5 The Completeness Theorem: Information describing a node reaches all other nodes in the system, under rule 4.17.

Proof: the first terms of rule 4.17 advertises a node to all its neighbors. The second and third terms of these rules propagate the information from one link to all other links. Therefore, each system is advertised to its neighbors, and because of the imposed spanning tree, the advertised information is propagated to all other nodes within the system. □

THEOREM 4.6 The Correctness Theorem: No system's attributes appear in any update vector more than once using rule 4.17.

Proof: Assume that there exists some y that appears more than once in an update vector for a node x. Under rule 4.17, only at a node y does the description information for y enter the system; all other nodes only propagate the information. Therefore, either $y = x$ and x appears in one of the incoming update vectors, or y appears in multiple incoming update vectors. The first indicates a cycle in the update vectors, and the second indicates two paths between x and y. Neither of these can occur in the presence of a spanning tree for a directed acyclic graph. Therefore, y cannot appear twice, and the theorem is proved. □

Theorems 4.5 and 4.6 demonstrate that the general semantics shown here are correct and globally complete.

4.6 Structuring Heuristics and Implications

Given a collection of machines, the question arises of how to best impose a distributed system structure using the rules defined in this chapter.

The tree-structured rules are the simplest, require the least overhead, and apply to a majority of existing administrative domains where organizations do not collaborate to manage a common resource pool. The typical case for a small department, in which all machines are within a single domain, can be accommodated by making all the machines children of single virtual node.

The primary-parent rules are appropriate for administering shared resources, if a viable spanning tree can be determined. If no viable spanning tree can be determined, the general rules will provide global completeness for any administrative structure.

If the graph becomes partitioned, either because of the failure of a communication link or because a host becomes unavailable, the mechanisms can automatically recover through the use of input timeouts. After a policy-specific number of missed input timeouts, a module will mark its neighbor as unreachable and will not incorporate the update vector for that neighbor into its outgoing description vectors until a new update vector is received from the neighbor.

There are three factors affecting the delay between a partitioning of the communications graph and when a node's incoming description vectors reflect the new system state: the time it takes for the neighbor nearest the failed node or link to mark the system as unreachable, the distance along the update path to the failed node or link, and the update frequency of other nodes along that path. Requests may be misdirected while the representation at each node is adapting to the new system state. This can result in a loss of efficiency, but will not produce erroneous scheduling results.

4.7 Summary

This chapter described three formal models for system update dissemination in distributed, hierarchical, autonomous systems. The first model is correct and globally complete for systems based on trees. The second model is correct and locally complete for a subclass of DAGs in which nodes have at most one common ancestor. The third model is correct and globally complete for DAGs with at most one path between nodes.

5. MESSIAHS: A PROTOTYPE IMPLEMENTATION

This chapter discusses a prototype implementation of the scheduling support mechanisms defined in chapters 3 and 4. The prototype implementation is called MESSIAHS: **M**echanisms **E**ffecting **S**cheduling **S**upport **I**n **A**utonomous, **H**eterogeneous **S**ystems.

MESSIAHS demonstrates how the abstract design set forth in chapter 3 can be mapped onto real architectures, and serves as a testbed for evaluating the design. The architectural model is machine-independent, and the prototype implementation uses SunOS 4.1 running on Sun-3 and SPARC workstations. Experimental results and analysis derived from executing several algorithms from the literature appears in chapter 6.

The first section describes the machine-dependent layer of the module, and the second section details the management layer. The third and fourth sections present two interface layers: one layer based on a library of function calls suitable for linking with a scheduler written in a high-level language, and another layer consisting of a policy specification language.

5.1 The Machine-Dependent Layer

The machine-dependent layer provides the interface in table 5.1 to the management layer of the module. The prototype does not implement those functions marked with a †.

The functions divide into three main groups: data collection, message passing, and task management. The data collection routines gather information that forms the system description for the local host. The message-passing routines implement

Table 5.1 Functions in the machine-dependent layer

Purpose	Function Name	Description
data collection	collect_process_data	collect data regarding the number of processes and load statistics
	collect_memory_data	collect data on available and total memory
	collect_disk_data	collect data on available temporary disk space
	collect_network_data	collect data on inter-module communication time
message passing	get_message	receive a message from the network
	send_message	send a message over the network
task management	suspend_task	pause a running task
	resume_task	continue executing a suspend task
	kill_task	halt execution of a task and remove it from the system
	checkpoint_task†	save the state of a task
	migrate_task†	checkpoints a task and moves it to a target host
	revert_task†	returns a task to its originating system

abstract message exchange between modules. The task management routines provide access to the underlying operating system process manipulation primitives.

The data collection operations are implemented using the kvm_open(), kvm_read(), kvm_nlist(), and kvm_close() routines that access kernel state in SunOS 4.1. The collect_process_data() function collects information on the number of processes in the ready queue, and the percentage of processor utilization. collect_memory_data() determines how much of the physical memory is in use. collect_disk_data() finds the amount of public free space on a system, typically in the /tmp directory on SunOS. collect_network_data() determines the average round-trip time between a host and its neighboring systems within the graph.

An alternative data collection implementation could use the rstat() call, which uses the Remote Procedure Call (RPC) mechanisms of SunOS to query a daemon that monitors the kernel state. However, the rstatd daemon does not provide information on physical memory statistics or communication time estimates, which are required to implement the mechanisms. Use of rstat() and rstatd also involves communication and context-switching overhead.

The message passing routines use the SunOS socket abstraction for communication and the User Datagram Protocol (UDP) to exchange information between modules. UDP was chosen because it provides an unreliable datagram protocol, which is the minimum level of service required for the update and control channels. The message passing routines encode the data using the XDR standard for external data representation.

The task manipulation primitives use the SunOS kill() system call, which sends a software interrupt, called a *signal*, to a process. The signals used are SIGSTOP, which pauses a process, SIGCONT, which resumes a paused process, and SIGKILL, which terminates a process. The task migration primitive is not implemented in the prototype, but is a stub procedure for later completion.

5.2 Abstract Data and Communication Management

The middle layer in figure 3.3 comprises the abstract data and task manipulation functions. These functions use the basic mechanism provided by the machine-dependent layer to construct higher-level semantic operations. For example, the send_sr() routine, which sends a schedule request to a neighbor, is implemented using the send_message() function. Table 5.2 lists the abstract data and task management functions.

The message-passing functions construct a message from the pertinent data and use the send_message() function to communicate with a neighboring module. There is one send routine for each message type defined in chapter 3.

MESSIAHS maintains two hash tables containing description vectors: one table containing description vectors of foreign tasks executing on the local host and another

Table 5.2 Functions in the abtract data and communication layer

Purpose	Function Name	Description
data exchange	send_sr	send a schedule request message
	send_sa	send a schedule accept message
	send_sd	send a schedule deny message
	send_trq	send a task request message
	send_ta	send a task accept message
	send_td	send a task deny message
	send_trv	send a task revoke message
	send_ssq	send a system status query
	send_ssv	send a system status vector
	send_tsq	send a task status query
	send_tsv	send a task status vector
	send_jr	send a join request
	send_jd	send a join deny
description vector access	sys_lookup	find the SDV for a system in the system hash table
	sys_first	return the first neighbor from the system hash table
	sys_next	return the next neighbor from the system hash table
	task_lookup	find the TDV for a task in the task hash table
	task_first	return the first task from the task hash table
	task_next	return the next task from the task hash table
events	register_event	insert an event into the timeout event queue
	enqueue_event	enqueue an event
	dequeue_event	dequeue an event
	new_queue	allocate an event queue
	qempty	check if a queue is empty
	set_input_timeout	enqueue an input timeout
	set_output_timeout	enqueue an output timeout
	set_recalc_timeout	enqueue a recalculation timeout
	set_revoke_timeout	enqueue a revocation timeout
	set_oto_period	set the output timeout period
	set_ito_period	set the input timeout period
	set_rcto_period	set the recalculation timeout period
	set_rvto_period	set the revocation timeout period

table for description vectors of neighboring systems. The hash tables use double hashing as described in Knuth [Knu73, pp. 521–526] for efficiency. The sys_lookup() and task_lookup() routines search the tables for a particular task or system. The sys_first(), sys_next(), task_first(), and task_next() routines iterate over the tables, returning successive description vectors with each call.

The event manipulation routines provide access to the internal event queues used by the module. The register_event() function inserts a timed event into the timeout queue, and the enqueue() and dequeue() routines allow direct manipulation of the queues. The set timeout routines enqueue timeout events of particular types, and the set period functions set the timeout periods for the various timers in MESSIAHS. If a timeout period is set to 0, the associated timer is disabled. Input timeouts occur when a neighbor has not sent a status message to the local host within the timeout period. Output timeouts indicate that the local host should advertise its state to its neighbors. Recalculation timeouts cause the local host to recompute its update vectors. When a revocation timeout occurs, the host checks its state to see if tasks should be revoked.

5.3 A Language for Policy Specification

This section describes a sample interface layer, called the MESSIAHS Interface Language (MIL). MIL is a policy specification language, and contains direct support for dynamic scheduling algorithms, without precluding support for static algorithms. Static algorithms consider only the system topography, not the state, when calculating the mapping. Dynamic algorithms take the current system state as input, and the resultant mapping depends on the state (see [CK88]). Figure 5.1 depicts the structure of an MIL program. The grammars for deriving the various rules, along with explanations of their semantics, appear in the rest of this section.

```
begin state
    <node state rules>
end
begin combining
    <data combination rules>
end
begin schedfilter
    <sched request filter rules>
end
begin taskfilter
    <task request filter rules>
end
begin revokefilter
    <revocation filter rules>
end
begin revokerules
    <revocation rules>
end
```

Figure 5.1 MIL specification template

5.3.1 Expressions and Types

MIL defines four basic types for data values: integers (INT), booleans (BOOL), floats (FLOAT), and strings (STRING). Integers are a sequence of decimal digits. Booleans have a value of either true or false. Floats are two decimal digit sequences separated by a decimal point, e.g. 123.45. Strings are a sequence of characters delimited by quotation marks (").

Identifiers are a dollar sign followed by either a single word or two words separated by a period. The latter case specifies fields within description vectors. The legal vectors are the received task description (task), the description of a task already executing on the system (loctask), the system description of a neighboring system (sys), the description of the local node (me), and the description being constructed by data combination (out). loctask is used to process task requests and revocation events. sys is used for the data combination rules and for schedule request. out is used only for the data combination rules, and me can appear in any of the six sections.

The following grammar defines the expression types used by the language. This grammar derives expressions of the base types only; in particular, there is no access to the Procclass field of the SDV with MIL.

$$
\begin{array}{lcl}
\textit{int-binop} & \rightarrow & + \mid - \mid / \mid * \mid \mathsf{mod} \mid \mathsf{\&} \mid \mathsf{|} \mid \mathsf{max} \mid \mathsf{min} \\
\textit{int-expr} & \rightarrow & \textit{int-expr int-binop int-expr} \mid \\
 & & (\textit{int-expr}) \mid \textit{integer} \mid \mathsf{int}(\textit{float-expr}) \mid \textit{id} \\
 & & \\
\textit{float-binop} & \rightarrow & + \mid - \mid / \mid * \mid \mathsf{max} \mid \mathsf{min} \\
\textit{float-expr} & \rightarrow & \textit{float-expr float-binop float-expr} \mid \\
 & & (\textit{float-expr}) \mid \textit{float} \mid \mathsf{float}(\textit{int-expr}) \mid \textit{id} \\
 & & \\
\textit{string-expr} & \rightarrow & \textit{string-expr} + \textit{string-expr} \mid \\
 & & (\textit{string-expr}) \mid \textit{string} \mid \textit{id} \\
 & & \\
\textit{comparator} & \rightarrow & < \mid > \mid == \mid >= \mid <= \mid <> \\
\textit{bool-binop} & \rightarrow & \mathsf{and} \mid \mathsf{or} \mid \mathsf{xor} \\
\textit{bool-expr} & \rightarrow & \textit{bool-expr bool-binop bool-expr} \mid \\
 & & \mathsf{not}\ \textit{bool-expr} \mid \\
 & & \textit{int-expr comparator int-expr} \mid \\
 & & \textit{float-expr comparator float-expr} \mid \\
 & & \textit{string-expr comparator string-expr} \mid \\
 & & \mathsf{match}(\textit{string-expr, string-expr}) \mid \\
 & & (\textit{bool-expr}) \mid \mathsf{true} \mid \mathsf{false} \mid \textit{id}
\end{array}
$$

5.3.2 Access to Intrinsic Mechanisms

MIL includes five task manipulation primitives: kill, suspend, wake, migrate, and revert. Other operations, such as process checkpointing, are available in the lower-level mechanisms, but are not explicitly included in the language. kill aborts a task, discards any interim results, and frees system resources used by the task. suspend temporarily blocks a running task. wake resumes a suspended task. migrate checkpoints a task and attempts to schedule the task on neighboring systems. revert checkpoints the task and returns the task to the originating system for rescheduling. Task revocation rules take the following form, using a boolean guard to determine when to take an action.

task-action	$\rightarrow$	kill |
		suspend |
		wake |
		migrate |
		revert
revocation-rule	$\rightarrow$	*bool-expr : task-action ;*

The node state section is a list of types, identifiers, and constant values. Node state declarations are parameters that affect system state. The four node state parameters are specint92, specfp92, recalc_timeout, and revocation_timeout. The specint92 and specfp92 parameters list the speed of the host in terms of the SPEC benchmarks [Staly]. The recalc_timeout and revocation_timeout parameters determine the timeout periods for the associated events.

5.3.3 Data Combination and Filters

MIL provides a mechanism to combine description vectors. To support communication autonomy, this mechanism allows the administrator to write rules specifying operations to coalesce the data.

int-action	$\rightarrow$	discard | set *int-expr*
float-action	$\rightarrow$	discard | set *float-expr*
bool-action	$\rightarrow$	discard | set *bool-expr*
string-action	$\rightarrow$	discard | set *string-expr*
combining-rule	$\rightarrow$	int *id bool-expr: int-action* ; |
		float *id bool-expr: float-action* ; |
		string *id bool-expr: string-action* ; |
		bool *id bool-expr: bool-action* ;

The boolean expression acts as a guard, and the action is performed for a particular *(type, identifier)* pair if the value of the guard is true. Administrators may supply multiple rules for the same pair. If multiple rules exist, the module evaluates them in the order written, performing the action corresponding to the first guard that evaluates to true.

If no matching rule is found for a pair, the identifier is discarded. Explicit discarding of data items, via the discard action, fulfills the constraint of communication autonomy. The set *value* action assigns *value* to the current pair in the outgoing description vector. An error in evaluating a guard automatically evaluates to false. If the evaluation of an action expression causes a run-time error, e.g. a division by 0, the action converts to discard.

The extension mechanism for description vectors allows the addition of simple attributes to the description of a system or task. The additional data must be of a primitive type; no aggregate types or arrays are permitted. The combining rules extend the description vectors, by adding new data fields with the set action.

In MIL, a filter is a series of guarded statements, similar to combining rules. In place of an *action*, filters define integer expressions,

$$\textit{filter-stmt} \quad \rightarrow \quad \textit{bool-expr} : \textit{int-expr} ;$$

A return value of 0 indicates that there is no match. A negative value indicates an error, and a positive value measures the affinity of the two vectors, with higher values indicate a better match. If the guard expression uses an undefined variable, the guard evaluates to false. If the integer expression references an undefined variable, the filter returns -1, indicating an error. With appropriate extension variables and guards, a single scheduling module can serve multiple scheduling policies.

5.3.4 Specification Evaluation

The extension and node state rules are interpreted when the specification is first loaded. The data combination rules are applied when a recalculation timeout occurs. When a revocation timeout occurs, the module passes once through the list of revocation rules, repeatedly evaluating each one until its guards return false. If the guard evaluates to true, the revocation filter is applied to the appropriate list of tasks to provide a target for the revocation action. If no task matches, the module moves on to the next rule in the list.

```
    begin state
1.      int     $recalc_period     60;
    end
    begin combining
2.      bool    $out.hasLaTeX    $sys.hasLaTeX:   set true;
3.      bool    $out.hasLaTeX    $sys.address == $me.address:
                                                  set true;
    end
    begin schedfilter
4.      $task.needsLaTeX and $sys.hasLaTeX and
            int($sys.loadave) < 5 :     6 - int($sys.loadave);
    end
```

Figure 5.2 A simple MIL specification

When a scheduling request arrives, the module iterates over the list of available systems, evaluating the request filter rules in order until a guard that evaluates to true is found, or the rules are exhausted. If no matching rule is found, 0 is returned. If a rule is found, its value is returned as the suitability ranking for that system. The module follows a similar procedure for task requests, iterating over the set of available tasks.

5.3.5 A Small Example

Figure 5.2 shows a simple MIL specification for a SPARC IPC participating in a distributed LaTeX text-processing system. Line 1 in the node state section sets the period for SDV recalculation at 60 seconds. Every minute, each system using this policy specification will compute its SDV and forward updates to its neighbors.

The SDV extension variable hasLaTeX is true if the system has LaTeX available and wishes to act as a formatting server. Clients requesting LaTeX processing set the needsLaTeX variable to true in their task description vector. The combining rule in line 2 sets the outgoing hasLaTeX variable if any of the incoming description vectors have it set, and the rule on line 3 sets the hasLaTeX variable for the local hosts. Hosts providing the LaTeX service would use line 3; hosts not providing the service would use line 2 to propagate advertisements by other hosts.

The scheduling filter rule in line 4 compares the available system vectors to the incoming task vector, accepts servers with load averages of less than five, and ranks the systems based on their load average. The guard would fail for a neighbor that has not set the hasLaTeX variable, and return false.

5.4 Example Algorithms

This section presents two applications built using MIL, in addition to the simple LaTeX batch processing system described earlier. The first application demonstrates the task revocation facility as used by a general-purpose distributed batch system. The second application implements a load-balancing algorithm.

5.4.1 Distributed Batch

The MITRE distributed batch [GSS89], Condor [BLL92], and Remote Unix [Lit87] systems support general-purpose distributed processing for machines running the UNIX operating system.

Recall that Condor has some support for execution autonomy. In particular, Condor includes a limited policy expression mechanism, with predefined variables and functions, without the possibility of extending the system or task description. Condor uses over 100 predefined variables and functions, all of which can be duplicated in MIL. Thus, MIL and Condor are roughly equivalent in terms of the effort required to customize the scheduling policy, but MIL and MESSIAHS provide additional autonomy support and extensibility.

Figure 5.3 lists a short specification file for a SPARC IPC participating in a distributed batching system. The state rules (lines 1–4) give the speed ratings for an IPC and the recalculation and revocation timeout periods.

```
    begin state
1.      float $SPECint92          13.8;
2.      float $SPECfp92           11.1;
3.      int   $recalc_period      30;
4.      int   $revocation_period  30;
    end
    begin combining
5.      string  $out.proctype  not match($out.proctype, "SPARC"):
                           set $out.proctype + ":SPARC";
6.      string  $out.OSname  not match($out.OSname, "SunOS4.1"):
                           set $out.OSname + ":SunOS4.1";
7.      string  $out.proctype  not match($out.proctype, $sys.proctype):
                           set $out.proctype + $sys.proctype;
8.      string  $out.OSname  not match($out.OSname, $sys.OSname):
                           set $out.OSname + $sys.OSname;
    end
    begin schedfilter
9.      $sys.address == $me.address and
             match($sys.proctype, $task.proctype) and
             match($sys.OSname, $task.OSname):
                          max(2000 - (1000 * int($sys.loadave)), 0);
10.     match($sys.proctype, $task.proctype) and
             match($sys.OSname, $task.OSname):
                          max(4000 - (1000 * int($sys.loadave)), 0);
    end
    begin revokefilter
11.     true:   1;
    end
    begin revokerules
12.     $me.loadave > 2.0 and $me.nactivetasks > 2:      suspend;
13.     $me.loadave < 1.0 and $me.nsuspendedtasks > 0:   wake;
    end
```

Figure 5.3 Remote execution specification

The combining rules in lines 5 and 6 ensure that the processor type variable, proc-type, contains the string ":SPARC" and that the operating system variable OSname contains the string ":SunOS4.1". Lines 7 and 8 propagate incoming processor and operating system names.

The example schedule request filter (lines 9 and 10) computes a rating function in the range [0, 2000] for the local system, and [0, 4000] for remote systems. The scheduling request rules ensure that the processor type and operating system match, and assign a priority to a match based on the system load average. Because there is no provision for requesting tasks from a busy system, the section for task request rules is empty.

Hosts participating in the batch system preserve autonomy by varying the parameters of the schedule request filter. For example, tasks submitted by a local user can be given higher priority by basing the rating function on the source address of the task.

The task revocation rules (lines 12 and 13) determine, based on the computational load on the node, whether active tasks should be suspended, or whether suspended tasks should be returned to execution. The `true` guard in the revocation filter rule (line 10) matches any available task, and the value portion of the rule assigns an equal priority to all tasks under consideration.

5.4.2 Load Balancing

Several researchers have investigated load balancing and sharing policies for distributed systems, such as those described in [Cho90], [ELZ85], and [Puc88].

The *Greedy Load-Sharing Algorithm* [Cho90], makes decisions based on a local optimum. When a user submits a task for execution, the receiving system attempts to place the task with a less busy neighbor, according to a weighting function. If no suitable neighbor is found, the task is accepted for local execution.

The suggested weighting function to determine if a task should be placed remotely is $f(n) = n \ div \ 3$, where n is the number of tasks currently executing on the local

```
    begin state
1.      int $recalc_period 5;
    end
    begin combining
2.      int $out.minload ($sys.address == $me.address) :
                     set min($out.minload, $me.ntasks);
3.      int $out.minload true:
                     set min($out.minload, $sys.minload);
    end
    begin schedfilter
4.      $sys.address == $me.address      : 1;
5.      $sys.minload <= ($me.ntasks / 3) : max(100 - $sys.minload, 2);
    end
```

Figure 5.4 Specification for Greedy Load Sharing

system. The algorithm searches for neighbors whose advertised load is less than or equal to one-third the local load. Because the Greedy algorithm depends on local state, it is dynamic.

The policy specification in figure 5.4 implements a variant of the Greedy algorithm. The original algorithm used a limited probing strategy to collect the set of candidates for task reception. The version in figure 5.4 sets the recalculation and retransmission periods low (line 1), and depends on the SDV dissemination mechanism to determine the candidate systems.

The combination rules (lines 2 and 3) set the `$minload` field to be the minimum of the load advertised by neighbors and the local load. The filter assigns a low priority to local execution (line 4), and rates the neighboring systems on a scale of two through 100 (line 5). Any eligible neighbor takes precedence over local execution, but if the resultant candidate set is empty, the local system executes the task.

The Greedy algorithm has no provision for task revocation; any tasks accepted run to completion. Thus, systems using the depicted specification yield some execution autonomy in the spirit of cooperation.

5.5 A Library of Function Calls

This section describes a library of function calls, called a *scheduling toolkit* that provides access to the underlying mechanism. The toolkit consists of the functions detailed in sections 5.1 and 5.2 combined with the functions listed in table 5.3.

Table 5.3 Functions in the MESSIAHS toolkit

Purpose	Function Name	Description
data exchange	send_Uvec	send the U update vector to a parent
	send_Dvec	send the D update vector to a child
	send_Svec	send the U update vector to a sibling
description vector manipulation	merge_SDV	merges two SDVs into one
	merge_statvec	merge two statistics vectors into one
	merge_procclass	merge two procclass sets into one
miscellaneous	mk_sid_sb	return a printable form of the system identification number
	Log	produce output in the error log
	pLog	produce output in the error log, including operating-system specific error messages

The send_Uvec(), send_Dvec(), and send_Svec() functions send update vectors to a system's parents, children, and siblings, respectively.

As shown in figure 5.5, statistics vectors (statvec) are components of the procclass structure, which are used to condense the advertised state information for a virtual system. Processors are grouped into *process classes* on a logarithmic scale, based on their computation speed. The statvec fields represent multiple processors using statistical descriptions of their capabilities. Processor speed was chosen as the grouping factor because research of the existing scheduling algorithms indicates that processor speed is the primary consideration for task placement (see chapter 2). The SPEC ratings were chosen as the default speed rating because they are the most widely available benchmark for both integer and floating point performance. Other measures of speed can be included through the extension mechanism.

The merge_statvec() function merges two statistics vectors, and merge_procclass() merges two processor classes into one. The merge_SDV() function provides a default mechanism for merging two system description vectors into one. The functions in figure 5.3 form the basis for MIL, described in section 5.3.

Table 5.4 Predefined event handlers in MESSIAHS

Function Name	Corresponding Event
handle_msg_sr	sched request message
handle_msg_sa	sched accept message
handle_msg_sd	sched deny message
handle_msg_trq	task request message
handle_msg_ta	task accept message
handle_msg_td	task deny message
handle_msg_trv	task revoke message
handle_msg_ssq	system status query message
handle_msg_ssv	system status vector message
handle_msg_tsq	task status query message
handle_msg_tsv	task status vector message
handle_msg_jr	join request message
handle_msg_jd	join deny message
handle_input_timeout	input timeout
handle_output_timeout	output timeout
handle_recalc_timeout	recalculation timeout
handle_revoke_timeout	revocation timeout

```
struct statvec {
    float min, max, mean, stddev, total;
};

typedef struct statvec Statvec;

struct procclass {
    bit32    nsys;      /* # of  machines in this class      */
    Statvec  qlen;      /* run queue statistics              */
    Statvec  busy;      /* load on cpu (percentage)          */
    Statvec  physmem;   /* total physical memory             */
    Statvec  freemem;   /* available memory                  */
    Statvec  specint92; /* ratings for SPECint 92            */
    Statvec  specfp92;  /* ratings for SPECfp 92             */
    Statvec  freedisk;  /* public disk space (/tmp) stats    */
};

typedef struct procclass Procclass;

#define SDV_NPROCCLASS 7
#define SDV_MAXUSERDEF  2048 /* multiple of 2 for cksum      */

struct SDV {
  SysId     sid;             /* Autonomous System ID         */
  bit32     nsys;            /* number of total systems      */
  bit32     ntasks;          /* number of total tasks        */
  bit32     nactivetasks;    /* number of active tasks       */
  bit32     nsuspendedtasks; /* number of suspended tasks    */
  float     willingness;     /* probability of taking on     */
                             /* a new task                   */
  float     global_load;     /* global load average          */
  Procclass procs[SDV_NPROCCLASS]; /* information on the     */
                             /* different classes of procs   */
                             /* in the autonomous system     */
  bit32     userdeflen;      /* length of user-defined data  */
  bit8      userdef[SDV_MAXUSERDEF]; /* user defined data    */
};

typedef struct SDV Sdv;
```

Figure 5.5 MESSIAHS data structures

```
nt = 0;
for (i = 0; i < SDV_NPROCCLASS; i++) {
     pp = &(psdv->procs[i]);
     if (pp->nsys > 0) {
          float ps, la, d;

          la = pp->qlen.min;
          ps = pp->specint92.mean;
          d = (la + 1) * ptdv->runtime * ptdv->specint92;
          value = (int) (1000 * ps / d);
          if (value > nt) {
               nt = value;
          }
     }
}
return nt;
```

Figure 5.6 Toolkit implementation of the ABS algorithm

The programmer uses the toolkit to write a set of event handlers, as discussed in section 3.2.2.1. These handlers comprise the scheduling policy. MESSIAHS predefines the set of handlers listed in table 5.4, which may be overloaded by the administrator to create a new policy.

As an example, the MESSIAHS prototype includes a default handler for schedule request messages. The administrator customizes the scheduling policy by writing a filter routine. Figure 5.6 lists the code for Arrival Balanced Scheduling [Bla92], figure 5.7 lists the code for the greedy algorithm, and figure 5.8 lists the code for the BOS algorithm. The next chapter analyzes the performance of these implementations.

The implementations of three algorithms demonstrate that the underlying mechanisms are easy to use. The longest of the three algorithms, BOS, represents less than one-half of one percent of the code for the scheduling support module. Writing a new algorithm involves editing a code skeleton and inserting the algorithm code in a C `switch` statement. This process takes only a few minutes for a programmer familiar

```
if (sidmatch(psdv->sid, pmysdv->sid)) {
     return 1;
} else if(psdv->global_load <= (pmysdv->ntasks / 3)) {
     gl = (int) psdv->global_load;
     nt = (int) psdv->ntasks;
     value = ((100 - gl) * 1000) + (999 - nt);
     return(value);
} else {
     return 0;
}
```

Figure 5.7 Toolkit implementation of the greedy algorithm

with the MESSIAHS code. In contrast, writing a scheduler from scratch, including data collection, data communication, and task management would take man-months of effort.

This ratio of schedule code size to support code size is consistent with that seen in other distributed scheduling support systems, such as Condor. However, MESSIAHS has ease-of-use advantages because of its separation of mechanism and policy, and because of its support for customizable scheduling policies.

5.6 Summary

This chapter described the MESSIAHS prototype implementation of the scheduling support mechanisms discussed in chapter 3. The prototype implements the mechanisms while adhering to the design principles from chapter 1.

Two sample user interfaces were defined: the MESSIAHS Interface Language, which facilitates rapid prototyping of new algorithms, and the scheduler's toolkit, which provides full access to the underlying mechanisms. Three sample MIL algorithms were given, and three sample schedulers were developed using the toolkit.

```
i = 100000;

if (is_sibling(pste)) {
     return (0);
}

/* add in 'self' and 'target' */
if (pste == pmyste) {
     i -= ((pmysdv->ntasks + 1) * (pmysdv->ntasks + 1));
} else {
     i -= (pmysdv->ntasks * pmysdv->ntasks);
     gl = (pste->sdv.ntasks + 1) * (pste->sdv.ntasks + 1);
     nt = MAX(pste->sdv.nsys, 1);
     i -= (gl / nt);
}

for (pshte = sys_first(); pshte != (Shte *) NULL;
     pshte = sys_next(pshte)) {

     if (pshte->entry != pste) {
          nt = MAX(pshte->entry->sdv.nsys, 1);
          gl = pshte->entry->sdv.ntasks *
               pshte->entry->sdv.ntasks;
          i -= (gl / nt);
     }
}

return i;
```

Figure 5.8 Toolkit implementation of the BOS algorithm

6. EXPERIMENTAL RESULTS

This chapter describes the results of experiments conducted to gain insight into the performance of the prototype implementation. Three algorithms from different subtrees of the taxonomy presented in section 2.3 were implemented using the MESSIAHS prototype. The execution time of each algorithm was recorded for a variety of conditions. Section 6.3 compares the results to best-case execution times to determine the overhead associated with the prototype implementation.

The experiments performed use simulated tasks, generated with a Poisson distribution. The tasks were simulated because there were no actual job traces available for distributed, autonomous systems. Section 6.2 gives the statistical basis for the simulated tasks and explains the derivation of the parameters for the experiments.

6.1 Experimental Architecture

The performance experiments use the system configuration listed in figure 6.1. The test configuration consists of six Sun 3/50 workstations and two Sun SPARC IPC workstations. The configuration is tree-structured, and includes two levels of encapsulating virtual systems. The longest path between nodes has three communication steps (e.g. maple—oak—ash—poplar; the oak—ash communication is directly between siblings, and bypasses elizabeth).

Each experiment embodies the following steps:

1. a scheduling module begins execution on each machine
2. a dispatcher program loops, performing the following three steps:
    (a) the next task description is loaded from the data file
    (b) the dispatcher sleeps until the delay period for the task expires

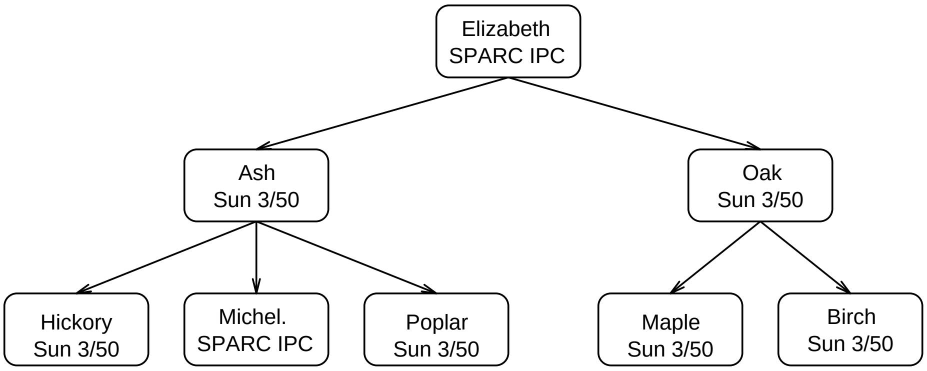

Figure 6.1 Experimental Configuration

(c) the dispatcher requests scheduling for the task

3. the simulation waits for all tasks to terminate

4. the scheduling modules cease execution

Each task records its execution information, including when it started and completed, in a log file.

The performance evaluation involved the execution of three distinct algorithms from the literature, and comparisons of their performance with a theoretical optimum. The algorithms were the Greedy Load-Sharing Algorithm (Greedy) from [Cho90], Arrival-Balanced Scheduling (ABS) from [Bla92], and a variation of the BOS algorithm (BOS) from [BJ91]. In all cases, the scheduling modules used an update frequency of one quanta.

The Greedy and ABS algorithms are dynamic, in that they schedule tasks as they arrive at the system. For each algorithm, task sets were generated with three different arrival rates and ten different means for execution times. The inter-task delay was selected from a uniform distribution between zero and the arrival rate parameter; e.g. for an inter-task delay parameter of three, tasks would arrive separated by zero, one, two, or three quanta with approximately equal frequency. This is the same model used in [Bla92].

The Greedy algorithm is classified as dynamic, distributed, non-cooperative, suboptimal, and heuristic. The Greedy algorithm rates the load on a host as the number of tasks the host has accepted for execution, without regard to processor speed. Figure 6.2 lists the decision filter for the Greedy algorithm. Self refers to the local host.

```
Rate self at 1;
For each neighbor 𝒩 {
        if (load on 𝒩 is ≤ one-third the local load) {
                rate 𝒩 at f(𝒩);
        }
}
Schedule the task on the highest-rated system;
```

Figure 6.2 Pseudocode for the Greedy algorithm

$f(\mathcal{N})$ depends on the minimum load of any host represented within $\mathcal{N}$'s status vector (minload) and the number of tasks running on all hosts encapsulated within $\mathcal{N}$ (ntasks). $f(\mathcal{N})$ is defined as

$$f(\mathcal{N}) = (100 - \mathcal{N}.minload) * 1000 + 999 - \mathcal{N}.ntasks.$$

$f(\mathcal{N})$ uses minload as its primary determinant of task placement, with the number of total tasks used to break ties.

The ABS algorithm uses more system description information, including the processor speed. The Casavant and Kuhl taxonomy classifies ABS as dynamic, distributed, cooperative, suboptimal, and heuristic. The pseudocode in figure 6.3 represents the implementation of the ABS algorithm.

The ABS algorithm estimates the execution time of a task for a processor, based on the computational load and speed of the processor, and uses that estimate to select a target system for scheduling.

For each execution-time mean, 96 tasks are selected from a Poisson distribution with a given mean (see 6.2 for details). The experiments use means in the range

```
for each system S, including self and all neighbors 𝒩 {
        nt = 0;
        for (each procclass PC in S's description vector) {
                if (PC contains at least one system) {
                        la = the lightest load in PC;
                        ps = the average speed of PC;
                        tr = estimated runtime of the task on the base processor;
                        d = (la + 1) * tr;
                        value = (int) (1000 * ps / d);
                        if (value > nt) then nt = value
                }
        }
        rate S at nt;
}
```

Figure 6.3 Pseudocode for Arrival-Balanced Scheduling

$\{1, 2, \ldots, 10\}$. Thus thirty tests are run for each of the Greedy and ABS algorithms. To facilitate reproducability and analysis, the same task set is used in both tests.

The BOS algorithm does not take inter-task delay into account, because it is a static, suboptimal, heuristic algorithm and requires that all tasks be available for placement at the same time. Therefore, the experiment using BOS employed ten test runs, one for each execution-time mean between one and 10. Figure 6.4 lists the BOS algorithm.

To determine the optimal performance for the algorithms, a program simulated the execution of each algorithm in the presence of perfect information describing the state for all eight hosts. This simulation was event-based, triggering an event each time a task was submitted or a task completed. No time was charged to a task for the computation of the scheduling policy; therefore, these best-case results are overly optimistic because they include only running time for the tasks. Each of the sample task sets was simulated for each algorithm. The results of these experiments appear in section 6.3.

```
for each task t {
    min = MAXINT;
    for each system S ∈ 𝒮 {
        temporarily assign t to S;
        c = Σ_{M∈𝒮} (M.ntasks)^2;
        if (c < min) {
            min = c;
            minsys = S;
        }
    }
    schedule t on minsys;
}
```

Figure 6.4 Pseudocode for the BOS algorithm

6.2 Statistical Background

As mentioned, the experiments use simulated tasks, generated with a Poisson distribution. A Poisson distribution is accepted as closely modeling job behavior in computer systems (see [Bla92, Fin88]). A Poisson distribution has a single parameter, μ, and is defined as

$$f(x) = \frac{\mu^x e^{-\mu}}{x!}, x = 0, 1, 2, \ldots \tag{6.1}$$

The parameter μ is both the mean and the variance of the distribution. Figure 6.5 displays Poisson distributions with μ values of 3, 5, and 10.

A result of the Central Limit Theorem (see [HPS71, chapter 7]) is that, for a continuous distribution with mean μ, and standard deviation σ, a sample S_n of size n, and an error limit c,

$$P\left(\left|\frac{S_n}{n} - \mu\right| \geq c\right) \approx 2(1 - \Phi(\delta)) \tag{6.2}$$

where

$$\delta = \frac{c\sqrt{n}}{\sigma} \tag{6.3}$$

and Φ is the normal distribution evaluated at δ.

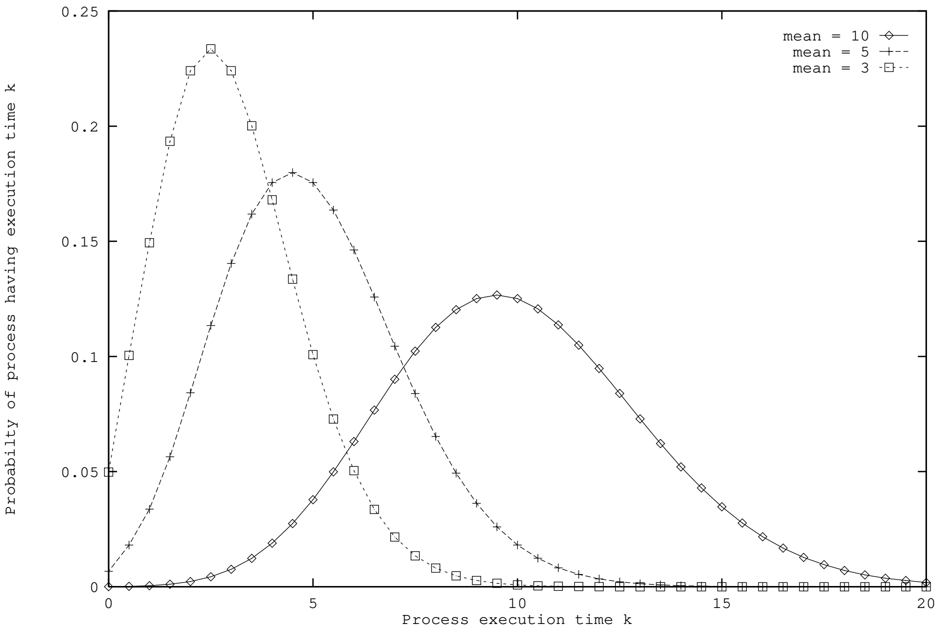

Figure 6.5 Sample Poisson distributions

The variance of a continuous distribution is the square of the standard deviation, so for the Poisson distribution, $\mu = \sigma^2$. Solving equation 6.3 for n yields

$$n = \frac{\delta^2 \sigma^2}{c^2} = \frac{\delta^2 \mu}{c^2} \tag{6.4}$$

To apply equations 6.2 and 6.3, the value of the error limit, c, and the confidence interval must be chosen. The smallest measurable time in the simulations was one-fifth of a quanta. To facilitate scalability, and because of computational limits of the simulation, the error limit was made relative to the mean of the distribution, and set at $c = .2\sqrt{\mu}$.

Substituting the values of μ and c in equation 6.3 yields

$$n = \frac{\delta^2 \mu}{.04\mu} = 25\delta^2.$$

Thus, equation 6.2 becomes

$$P\left(\left|\frac{S_n}{25\delta^2} - \mu\right| \geq .2\sqrt{\mu}\right) \approx 2(1 - \Phi(\delta)).$$

Setting the left hand side to a confidence interval of 95%, so that the probability on the left hand side equals 5%, yields

$$0.05 \approx 2(1 - \Phi(\delta)).$$

Solving for δ and consulting a table of values for the normal distribution yields $\delta = 1.96$, and thus

$$n = 25\delta^2 = 96.$$

Thus, with a sample size of 96 generated tasks, it is 95% probable that the sample mean differs from the theoretical mean by no more than $0.2\sqrt{\mu}$.

6.3 Results and Analysis

This section gives the results for each of the experiments and analyzes the results, comparing the results to the simulated performance of the algorithm in the presence of perfect information. The experiments estimate the computational overhead caused by using the MESSIAHS prototype. Two components comprise the overhead: overhead caused by computing the schedule, and overhead incurred because of inefficiency in the computed schedule. No attempt was made to separate the two types of overhead. It is important to realize that even if the algorithms compute an optimal schedule, the performance will still appear suboptimal because of the cost of computing the schedule, which does not appear in the theoretical minimum time.

During the execution of the experiments, the resource usage of the MESSIAHS module was measured. The module places a computational load of less than 0.5% of the CPU time on a Sun 3/50[1] when the module is exchanging update or control messages, or starting new tasks. The scheduling module uses between 68 and 416 kilobytes of memory. The module uses no disk space.

Results for each tested algorithm appear in tabular and graphical form. For the tables, the *Mean* column lists the execution-time mean and the *Delay* column, if it appears, lists the maximum inter-task delay. The *Minimum* column lists the

[1] As measured by the `top` program during the simulations.

theoretical minimum running time, and the *Actual* column lists the actual running time. Running time is measured from the time the first task is submitted to the system until the time the last task finishes execution. The *Overhead* column lists the difference between the Actual and the Minimum times, as a percentage of the Minimum[2].

The bar graphs display the information visually. For each test and execution-time mean, pairs of bars appear. The dark bar represents the minimum running time and the light bar represents the actual running time.

6.3.1 The Greedy and ABS Algorithms

The results for the Greedy and ABS algorithms appear together because they are both dynamic algorithms.

Table 6.1 displays the results using the Greedy algorithm. The Greedy algorithm runs within 7.12% of the optimum, on average, with a standard deviation of 7.38. Figures 6.6 and 6.7 show graphs of the minimum and measured execution times for the Greedy algorithm.

Table 6.2 lists the results for ABS, and figures 6.8 and 6.9 display the results in graphical form. The mean of the overhead is 9.67%, with a standard deviation of 8.09. Note that although the relative overhead is higher for ABS than for Greedy, the absolute running time is very nearly the same. The overhead for ABS appears higher because ABS has a lower theoretical minimum running time than does the Greedy algorithm.

The largest discrepancies for both the Greedy and ABS algorithms occur in the tests with inter-task delay of at most one quantum. This overhead occurs because the algorithms are using out-of-date information, and the ABS algorithm receives a heavier relative penalty than does Greedy because ABS is more sensitive to stale system description information. As the inter-task delay increases, updates percolate

[2]That is, Overhead $= \frac{\text{Actual} - \text{Minimum}}{\text{Minimum}}$.

Table 6.1 Experimental Results for the Greedy Load-Sharing Algorithm

Mean	Delay	Minimum	Actual	% Overhead
1	1	4.72	5.02	6.35
	2	8.18	8.55	4.52
	3	13.23	13.57	2.57
2	1	4.28	4.70	9.81
	2	9.88	10.32	4.45
	3	14.45	14.65	1.38
3	1	4.30	4.83	12.32
	2	8.88	9.17	3.27
	3	14.48	14.80	2.21
4	1	4.28	4.95	15.65
	2	10.30	10.73	4.17
	3	14.23	14.53	2.11
5	1	4.87	5.17	6.16
	2	10.33	10.75	4.07
	3	14.27	14.55	1.96
6	1	4.92	6.03	22.56
	2	9.82	10.02	2.04
	3	12.93	13.33	3.09
7	1	5.05	5.85	15.84
	2	9.85	10.10	2.54
	3	12.98	13.15	1.31
8	1	5.12	6.12	19.53
	2	10.93	11.18	2.29
	3	14.37	14.68	2.16
9	1	5.43	5.98	10.12
	2	10.98	11.63	5.92
	3	13.80	14.05	1.81
10	1	5.48	7.31	33.39
	2	8.98	9.48	5.57
	3	13.85	14.48	4.55

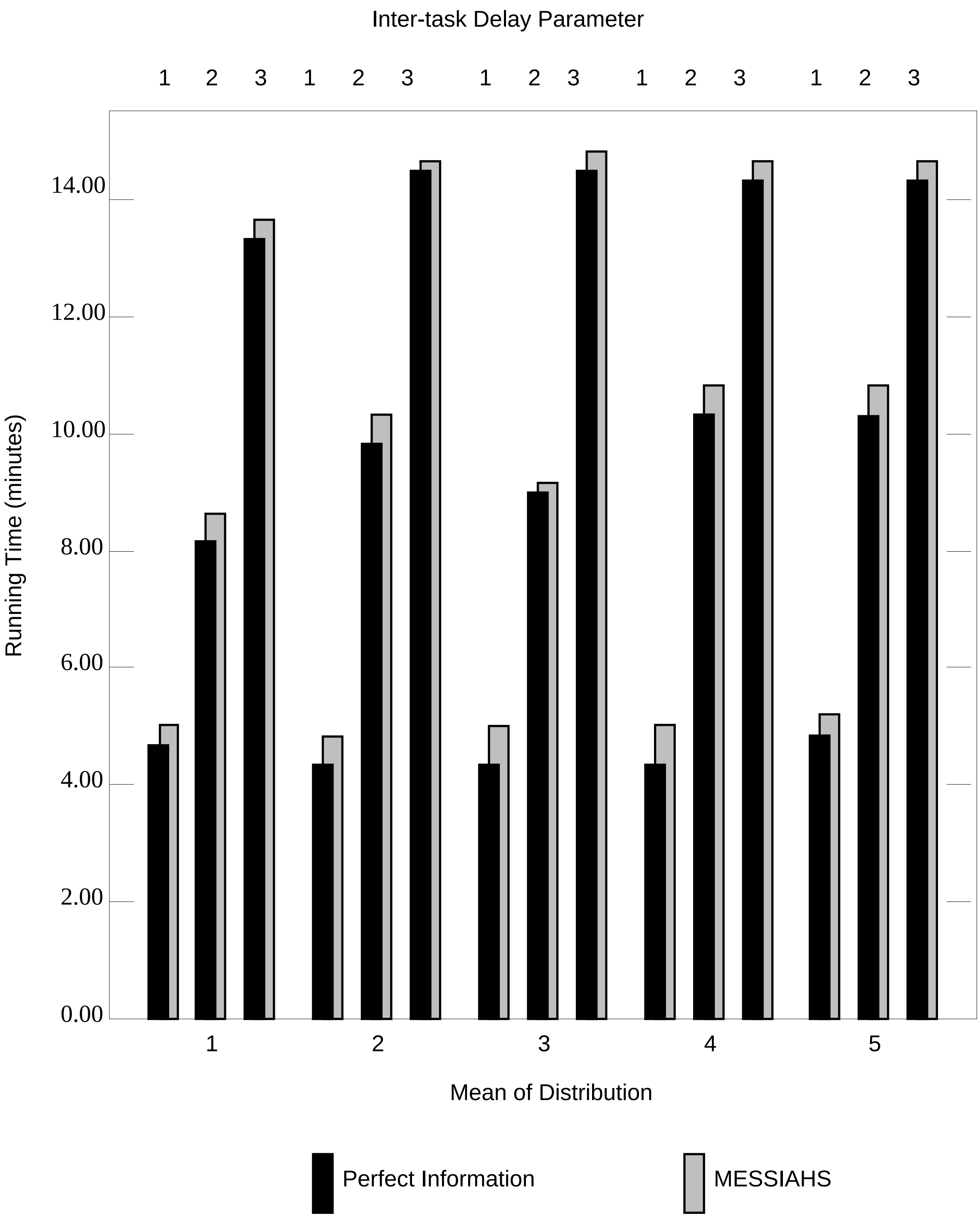

Figure 6.6 Performance of the Greedy algorithm with mean 1–5

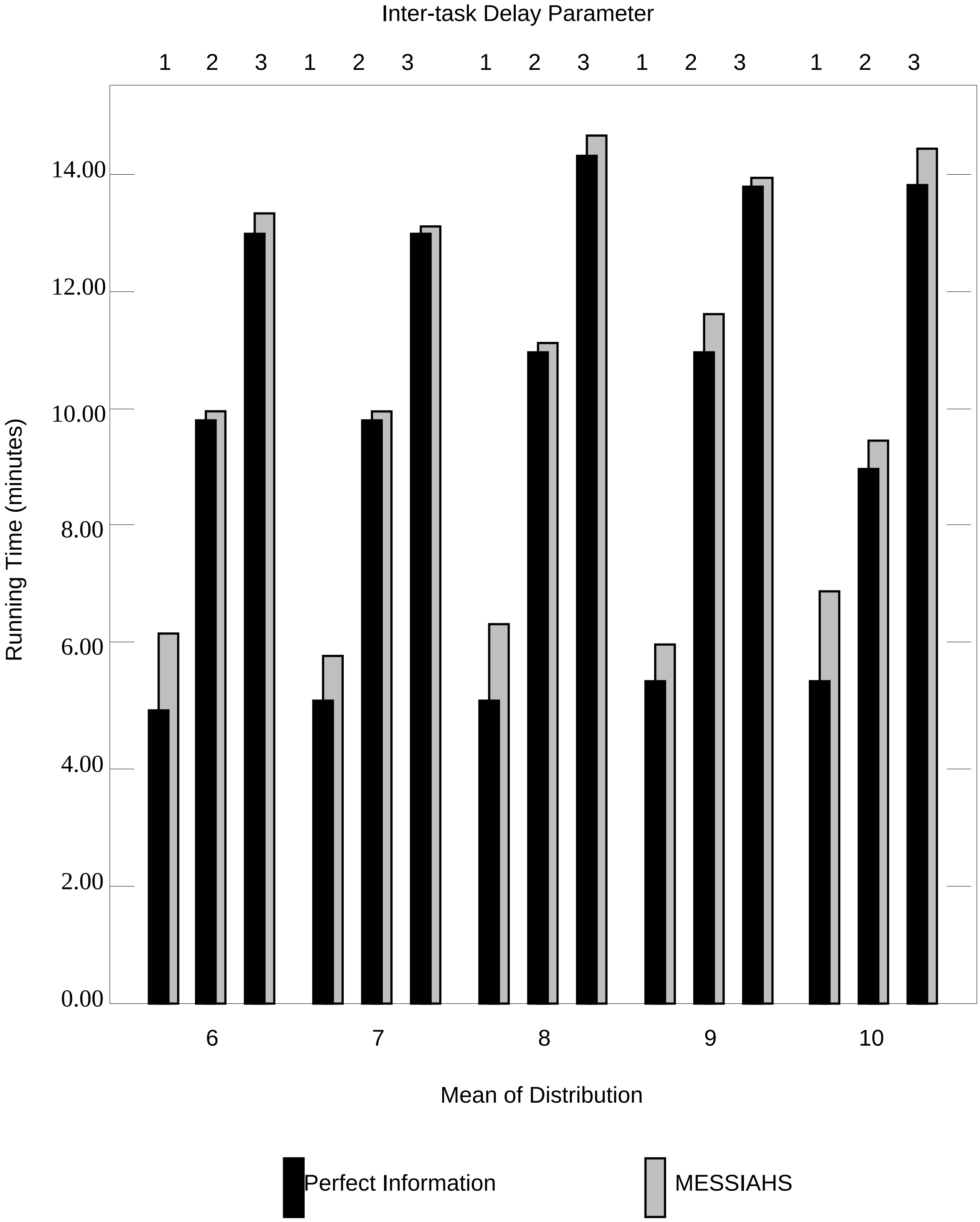

Figure 6.7 Performance of the Greedy algorithm with mean 6–10

Table 6.2 Experimental Results for Arrival Balanced Scheduling

Mean	Delay	Minimum	Actual	% Overhead
1	1	4.72	5.03	6.57
	2	8.12	8.58	5.67
	3	13.22	13.52	2.27
2	1	4.22	4.82	14.21
	2	9.82	10.15	3.36
	3	14.42	14.73	2.15
3	1	4.22	4.77	13.03
	2	8.73	8.98	2.86
	3	14.42	14.73	2.15
4	1	4.22	5.28	25.12
	2	10.13	10.80	6.61
	3	14.13	14.47	2.41
5	1	4.73	5.48	15.86
	2	10.15	10.90	7.39
	3	14.13	14.38	1.77
6	1	4.73	5.28	11.63
	2	9.57	9.78	2.19
	3	12.67	13.50	6.55
7	1	4.85	6.02	24.12
	2	9.57	10.07	5.22
	3	12.67	12.98	2.45
8	1	4.85	6.25	28.87
	2	10.58	11.67	10.30
	3	13.98	14.92	6.72
9	1	5.17	5.97	15.47
	2	10.60	11.70	10.38
	3	13.32	14.12	6.01
10	1	5.17	6.78	31.14
	2	8.52	9.45	10.92
	3	13.32	14.22	6.76

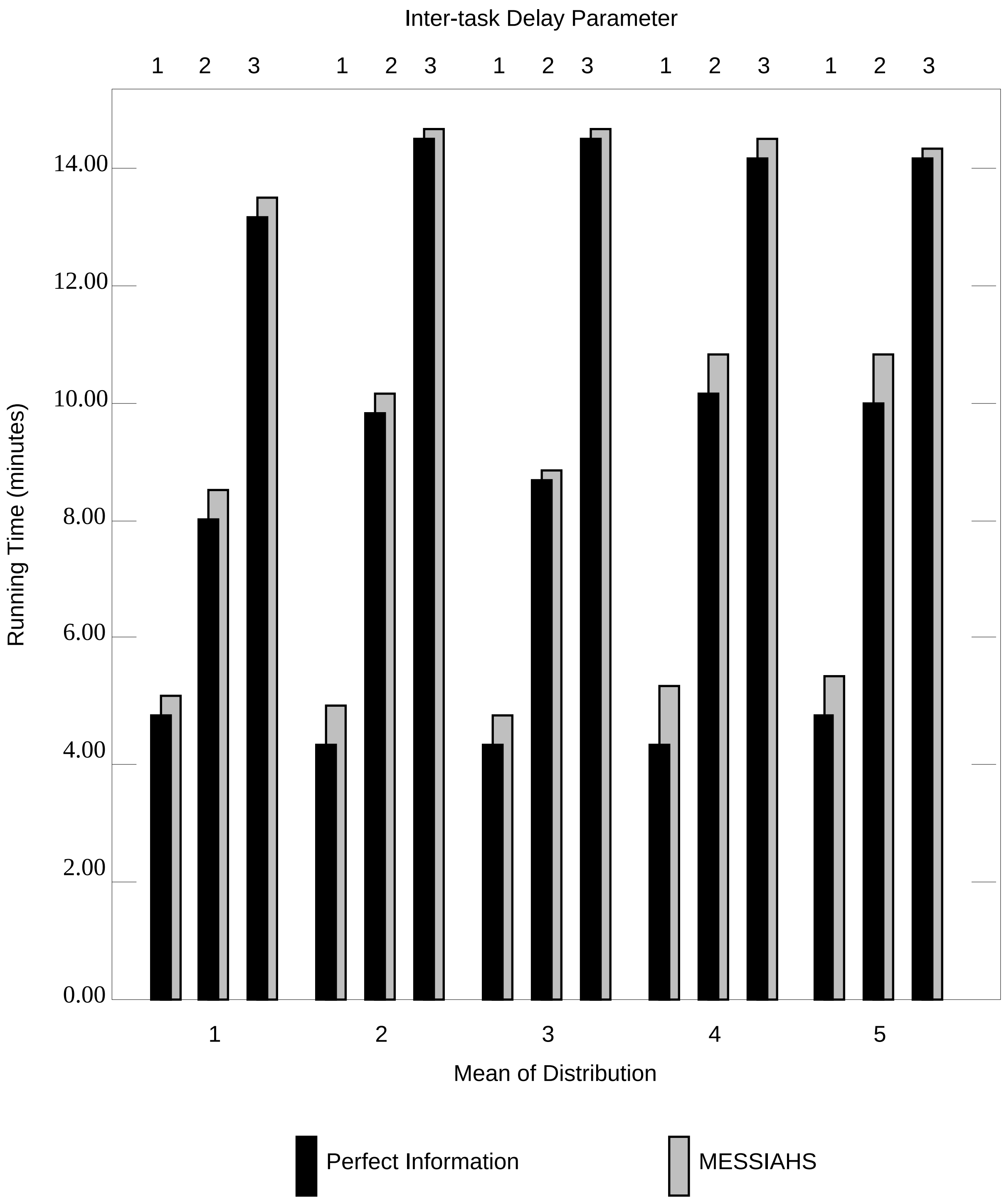

Figure 6.8 Performance of Arrival Balanced Scheduling with mean 1–5

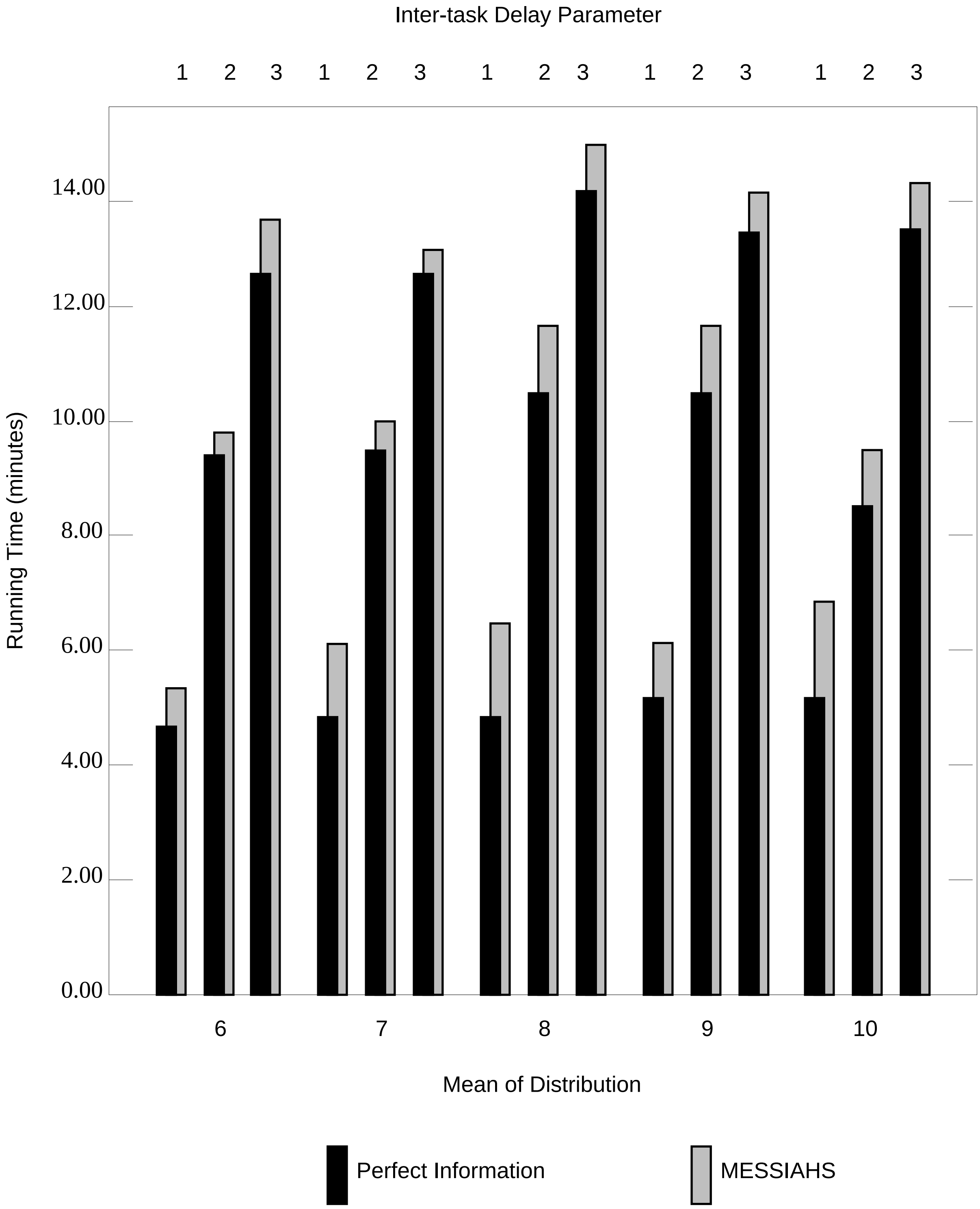

Figure 6.9 Performance of Arrival Balanced Scheduling with mean 6–10

further through the system between each task's arrival, so that the scheduling modules have more accurate description information with which to work.

6.3.2 The BOS Algorithm

The results for the BOS algorithm appear in table 6.3 and figure 6.10. The overhead for BOS has a mean of 36.53% and a standard deviation of 10.28. These numbers are much higher than for the dynamic algorithms. There are several reasons for this. The MESSIAHS mechanisms do not schedule task forces in a single pass; the tasks must be submitted and scheduled piecemeal. This causes delay, relative to the minimum, because the tasks are scheduled serially. In the minimum case, they are all scheduled simultaneously, with no passage of simulated time. In addition, the computational requirements of the BOS algorithm ($O(n^2)$) are higher than the requirements of ABS or Greedy (both are $O(n)$). Again, the minimum time estimate does not include this computational cost.

Table 6.3 Experimental Results for the BOS algorithm

Mean	Minimum	Actual	% Overhead
1	0.68	0.92	34.79
2	1.07	1.33	24.60
3	1.62	2.28	40.94
4	2.05	2.78	35.77
5	2.32	3.30	42.24
6	2.72	3.52	29.41
7	3.23	3.93	21.67
8	3.65	5.72	56.66
9	4.20	5.48	30.55
10	4.63	6.88	48.66

These causes of overhead will appear, in general, in any static algorithm that is run as a dynamic algorithm. Static algorithms have no run-time cost, because static algorithms devise the schedule at compile time. However, dynamic versions of static algorithms are applicable in a wider range of areas, because dynamic algorithms do not assume a static system description or configuration.

The implementation of a static algorithm provides an example of a worst-case baseline. The task assignments were made without any intervening state advertisement, so the results represent the performance of a scheduling policy in the absence of fresh data. This data point gives an insight into the lower bound of expected performance of scheduling algorithms using MESSIAHS, provided that complete and accurate information is initially provided. No conclusions can be drawn about the performance of scheduling algorithms in the presence of insufficient or widely inaccurate information.

6.4 Summary

This chapter presented the results of experiments run using three algorithms over the MESSIAHS mechanisms. The algorithms represented three different classes from the taxonomy presented in section 2.3. The implementation of three different algorithms demonstrates the feasibility and generality of the mechanisms.

The results indicate, but do not prove, that the overhead incurred by use of the prototype is minor, typically less than 10% for dynamic algorithms and less than 40% for static algorithms. The 40% slowdown for a static algorithm may be acceptable in some environments, because the MESSIAHS version of the algorithm works in an environment the original static algorithm could not.

In addition, it appears that the MESSIAHS mechanisms perform better as the ratio of inter-task delay to update frequency increases. This increased ratio means that update information travels farther within the distributed system between task arrivals, and thus the scheduling modules are working with more up-to-date information.

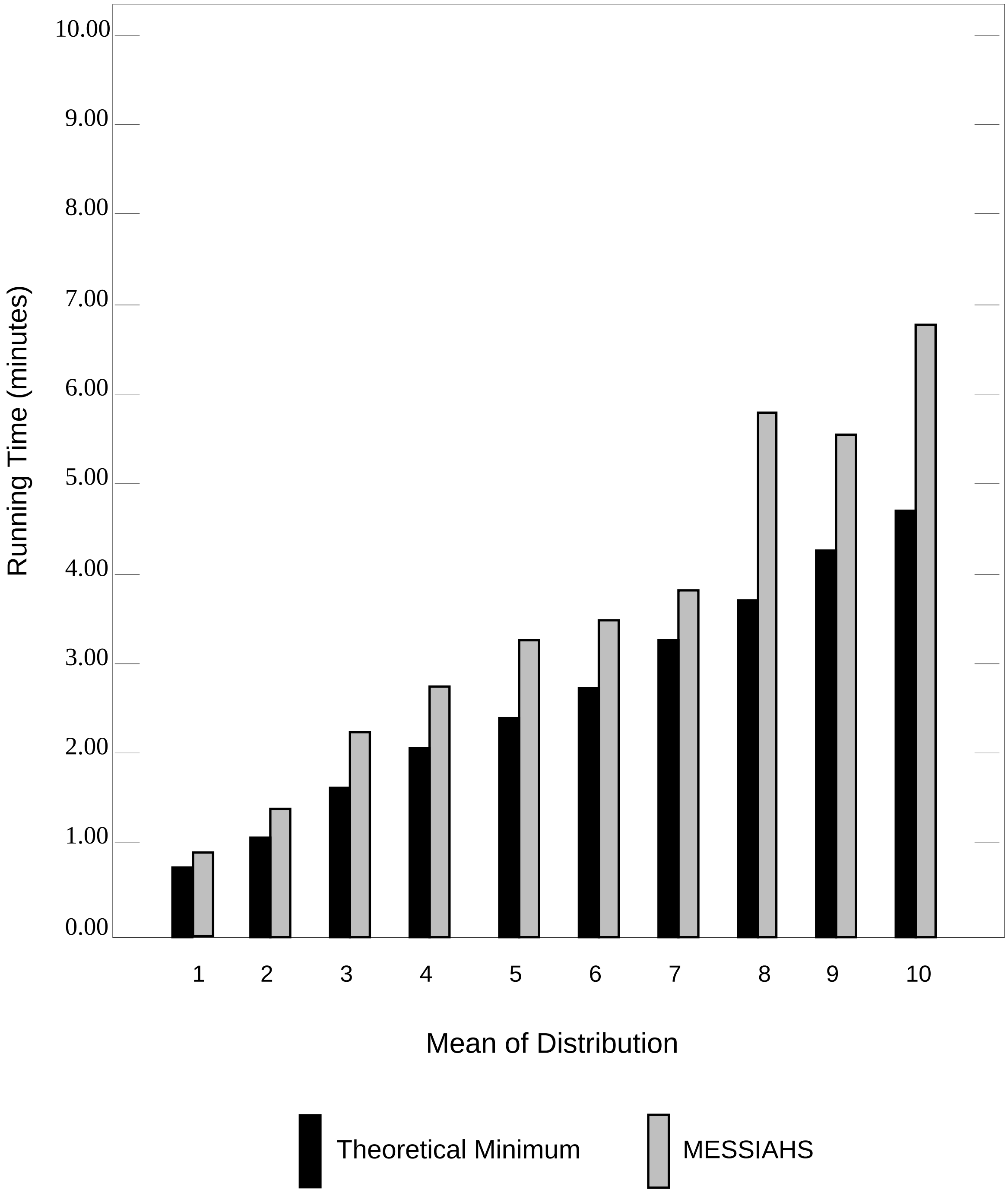

Figure 6.10 Performance of the BOS algorithm with mean 1–10

7. CONCLUSIONS AND FUTURE WORK

This dissertation has described the formulation, design, and implementation of scheduling support mechanisms for autonomous, heterogeneous, distributed systems. This chapter reviews the work presented in this dissertation, highlights the contributions made by this work, and projects directions for future research.

7.1 Design

The mechanisms presented in this dissertation provide automated support for task placement in distributed systems, while accommodating scalability, autonomy, efficiency, and extensibility. The centerpoint of the design is the virtual system.

Virtual systems represent a subset of the capabilities of one or more real machines. Distributed systems can be constructed by collecting virtual systems into encapsulating virtual systems, yielding a hierarchical overall structure. This structure corresponds to the organizational structure observed in existing administrative domains for computer systems. Each virtual system is implemented with a scheduling module that is responsible for local task management, communication with other virtual systems, and implementing the local scheduling policy.

Each host within the distributed system reserves the authority to make all policy decisions locally. No host is compelled to execute remote tasks. Each virtual system may have a distinct scheduling policy, which may operate in cooperation or in conflict with other policies. Parameters that affect system behavior, such as timeout periods, are set by the local administrator. This freedom to set local policy also allows administrators to use the mechanisms poorly.

The hierarchical nature of virtual systems localizes message traffic and allow the condensation of multiple system descriptions into a constant size. Decentralized information storage and control eliminates most single points of failure or bottlenecks that limit scalability.

To participate in the scheduling system, modules sacrifice some communication autonomy and exchange messages according to the communications protocols defined in chapter 3. To keep interference low, the mechanisms provide for either polled or timed updates. The information condensation mechanisms coalesce description vectors for the systems that comprise a virtual system into a single vector describing the encapsulating system, thus decreasing network traffic and message processing time. The update vectors include an extension feature that allows them to be tailored to new or specific scheduling algorithms.

As part of their support for generality, the mechanisms support heterogeneous systems. This support for heterogeneity includes the use of an external data representation, the ability to describe disparate architectures in a system description vector, and the ability to execute tasks best-suited to different architectures on the most appropriate machine.

Chapter 4 defined the concept of completeness and correctness for update vectors in distributed systems. Chapter 4 also contains proofs of correctness and completeness for information dissemination mechanisms in three different classes of distributed system architectures. The information dissemination mechanisms ensure that, in the absence of policy restrictions, a description of every host reaches every other host within the distributed system, and also precludes overestimation of available resources. Administrators can supply false information to the mechanisms, but cannot corrupt the mechanisms themselves.

7.2 Implementation

As part of the work presented here, we constructed the MESSIAHS prototype implementation of the scheduling support mechanisms.

7.2.1 Interface Layers

MESSIAHS includes a scheduling language, called the MESSIAHS Interface Language, which provides for rapid prototyping of new scheduling algorithms. A drawback of MIL is the computational overhead of the prototype implementation of the interpreter.

An alternative vehicle providing access to the underlying mechanisms is a library of high-level language functions called a scheduler's toolkit. MESSIAHS includes a toolkit that is more complex than MIL but yields policies with less run-time overhead. The toolkit is accessible from several high-level languages, and can be used by application programs to schedule tasks through the scheduling modules.

Both of these interface layers significantly ease the implementation of new scheduling policies. As shown in chapter 5, the time required to prototype new algorithms using MESSIAHS can be as little as a few minutes.

7.2.2 Experimental Results

Experimental implementations of three algorithms from the literature show that the prototype can yield efficient schedulers and efficient schedules. The efficient scheduler has low overhead, typically consuming less than 0.5% of the CPU resources of a machine. The resulting scheduler produces schedules that typically have a total running time of within 10% of the running time of a schedule produced with complete, up-to-date information.

The results from chapter 6 indicate that the overhead caused by MESSIAHS is acceptable for both static and dynamic algorithms. The MESSIAHS mechanisms perform better as the ratio of inter-task delay to update frequency increases, with a typical slowdown of less than ten percent. This means that MESSIAHS is useful in situations where the update frequency can be set to exceed the task submission frequency.

7.3 Contributions

The work described in this dissertation makes several contributions to scheduling support for distributed computing systems. These contributions were detailed in previous sections, and are recapitulated here.

- a general design and hierarchical architecture for scheduling support in distributed, autonomous systems
- support for existing scheduling algorithms
- extensible, flexible mechanisms to support future scheduling algorithms
- a prototype implementation of the mechanisms
- an abstract language for rapid prototyping and implementation of scheduling policies
- a flexible toolkit for constructing detailed scheduling algorithms
- performance data verifying the feasibility of this approach
- provably correct state dissemination methods for distributed systems

7.4 Future Work

Several areas of future work remain in the area of scheduling support. Some are extensions and completions of current MESSIAHS mechanisms, and others embody new directions for this research.

7.4.1 Extensions to MIL

At present, MIL does not allow the programmer to record history, so that adaptive algorithms cannot be written in MIL. Possible extensions to the MIL mechanisms include information persistence (history) and aggregate data types (records and arrays). In addition, substantial performance advantages could result from tuning of the interpreter to eliminate the excessive symbol table manipulation present in the current version.

As an alternative to MIL, other interface languages could be implemented. PERL [WS90] and Python [vR92] are interpreted languages with efficient implementations. Both are intended for combining traditional imperative programming languages, such as C [KR90], with command-line interpreters, such as the Bourne Shell [Bou]. Tcl [Ous93] is a command language that supplies a library of parsing functions that could be integrated into the MESSIAHS scheduling module in place of MIL. A PERL, Python, or Tcl interface language would allow information persistence and aggregate data types while retaining MIL's advantages of rapid prototyping and ease of use.

7.4.2 Dynamic Aspects

Although support for dynamic system structure exists as a basic element in the MESSIAHS mechanisms, the current implementation uses static configurations. Dynamic configuration permits administrators to specify scheduling policies that remove a host from the pool of available processors, and to rejoin at a later time. In the case of heavy loads, the system could reconfigure itself to better serve the application mix. Dynamic system configuration could be extended to support fault tolerance, so that the communications graph is rebuilt around a faulty node.

7.4.3 Task Migration

Checkpointing features and one of the task migration mechanisms described in chapter 2.1.2 will be incorporated into the MESSIAHS mechanisms.

7.4.4 Multi-Architecture Programs

To further support heterogeneity, some form of multi-architecture program binaries or architecture-independent program representation will be investigated.

7.4.5 Extensions to the Task Scheduling Mechanism

The prototype task scheduling mechanism schedules only one task at a time. The BOS experiment demonstrated the drawback of this approach for static schedulers, which have an entire task force available for scheduling at the same time. A future extension to the task scheduling mechanism will include the ability to group sets of tasks so that the entire group is scheduled simultaneously.

7.4.6 Security

The autonomy support present in MESSIAHS forms the basis for security mechanisms. A host can offer resources for consumption within the distributed system without revealing private configuration information. The autonomy support in the mechanisms enhances availability of resources and prevents denial-of-service attacks. Further exploration is planned in the areas of secure information advertisement, user and host authentication, and execution environment protection.

7.5 Summary

This dissertation examined new mechanisms for scheduling support in distributed systems. These mechanisms provide hierarchical structuring of computer systems and allow the conglomeration of disparate computers into a single distributed system. The system uses a decentralized architecture that supports autonomy in an efficient manner. A provably correct model of information dissemination provides a sound basis for part of the mechanisms, and ensures that complete and correct information reaches all hosts within the system.

The work presented here supports the thesis that efficient, extensible scheduling support mechanisms can be constructed for autonomous, heterogeneous, distributed systems. MESSIAHS incorporates these mechanisms into a prototype implementation that provides a scheduler's toolkit and a scheduling language. Implementation of existing scheduling policies using the toolkit shows that the mechanisms have low overhead and can facilitate the derivation of efficient schedules.

BIBLIOGRAPHY

BIBLIOGRAPHY

[ADD82] G. R. Andrews, D. P. Dobkin, and P. J. Downey. Distributed allocation with pools of servers. In *Proceedings of the Symposium on Principles of Distributed Computing*, pages 73–83. ACM, August 1982.

[AF87] Y. Artsy and R. Finkel. Simplicity, Efficiency, and Functionality in Designing a Process Migration Facility. In *The 2nd Israel Conference on Computer Systems*, May 1987.

[Agg93a] Distributed Builds: The Next Step in the Evolution of Programming Tools. Aggregate Computing, Inc., Minneapolis, MN, 1993.

[Agg93b] NetMake Technology Summary. Aggregate Computing, Inc., Minneapolis, MN, 1993.

[Agg93c] Using the NetShare SDK to Build a Distributed Application: A Technical Discussion. Aggregate Computing, Inc., Minneapolis, MN, 1993.

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, 1974. ISBN 0-201-00029-6.

[BF81] R. M. Bryant and R. A. Finkel. A stable distributed scheduling algorithm. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 314–323. IEEE, April 1981.

[BH91a] A. M. Bond and J. H. Hine. DRUMS: A Distributed Performance Information Service. In *14th Australian Computer Science Conference*, Sydney, Australia, February 1991.

[BH91b] A. M. Bond and J. H. Hine. DRUMS: A Distributed Statistical Server for STARS. In *Winter USENIX*, Dallas, Texas, January 1991.

[BJ87] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[BJ91] G. J. Bergmann and J. M. Jagadeesh. An MIMD Parallel Processing Programming System with Automatic Resource Allocation. In *Proceedings of the ISMM International Workshop on Parallel Computing*, pages 301–304, Trani, Italy, September 10–13 1991.

[BK90] F. Bonomi and A. Kumar. Adaptive Optimal Load Balancing in a Nonhomogeneous Multiserver System with a Central Job Scheduler. *IEEE Transactions on Computers*, 39(10):1232–1250, October 1990.

[Bla92] B. A. Blake. Assignment of Independent Tasks to Minimize Completion Time. *Software–Practice and Experience*, 22(9):723–734, September 1992.

[BLL92] A. Bricker, M. Litzkow, and M. Livny. Condor Technical Summary. Technical Report 1069, Department of Computer Science, University of Wisconsin-Madison, January 1992.

[BMK88] D. R. Boggs, J. C. Mogul, and C. A. Kent. Measured Capacity of an Ethernet: Myths and Reality. Technical Report 88/4, Digital Equipment Corporation, Western Research Laboratory, September 1988.

[Bon90] F. Bonomi. On Job Assignment for a Parallel System of Processor Sharing Queues. *IEEE Transactions on Computers*, 39(7):858–869, July 1990.

[Bon91] A. M. Bond. A Distributed Service Based on Migrating Servers. Technical Report CS-TR-91/4, Computer Science Department, Victoria University, Wellington, New Zealand, 1991.

[Bou] S. R. Bourne. *An Introduction to the UNIX Shell.* 4.3 BSD UNIX documentation.

[BPY90] M. Bowman, L. L. Peterson, and A. Yeatts. Univers: An Attribute-Based Name Server. *Software–Practice and Experience*, 20(4):403–424, April 1990.

[CA83] T. C. K. Chou and J. A. Abraham. Load redistribution under failure in distributed systems. *IEEE Transactions on Computers*, C-32(9):799–808, September 1983.

[Cas81] L. M. Casey. Decentralised scheduling. *Austrialian Computer Journal*, 13(2):58–63, May 1981.

[Che90] B. Cheswick. The Design of a Secure Internet Gateway. In *USENIX Summer Conference*, pages 233–237, June 1990.

[Cho90] S. Chowdhury. The Greedy Load Sharing Algorithm. *Journal of Parallel and Distributed Computing*, 9:93–99, 1990.

[CK79] Y. C. Chow and W. H. Kohler. Models for dynamic load balancing in a heterogeneous multiple processor system. *IEEE Transactions on Computers*, C-28(5):354–361, May 1979.

[CK84] T. L. Casavant and J. G. Kuhl. Design of a loosely-coupled distributed multiprocessing network. In *Proceedings of the International Conference on Parallel Processing*, pages 42–45. IEEE, August 1984.

[CK88] T. L. Casavant and J. G. Kuhl. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. *IEEE Transactions on Software Engineering*, 14(2):141–154, February 1988.

[CLL85] M. J. Carey, M. Livny, and H. Lu. Dynamic Task Allocation in a Distributed Database System. In *Distributed Computing Systems*, pages 282–291. IEEE, 1985.

[Com84] D. E. Comer. *Operating System Design: the XINU Approach*, volume 1. Prentice-Hall, 1984. ISBN 0-13-637539-1.

[Com91] D. E. Comer. *Internetworking with TCP/IP*, volume I, Principles, Protocols, and Architecture. Prentice Hall, second edition, 1991. ISBN 0-13-468505-9.

[Con58] M. E. Conway. Proposal for an UNCOL. *Communications of the ACM*, 1(3), 1958.

[CS92] S. J. Chapin and E. H. Spafford. Scheduling Support for an Internetwork of Heterogeneous, Autonomous Processors. Technical Report TR-92-006, Department of Computer Sciences, Purdue University, January 1992.

[CS93a] S. J. Chapin and E. H. Spafford. An Overview of the MESSIAHS Distributed Scheduling Support System. Technical Report TR-93-011 (supercedes TR-93-004), Department of Computer Sciences, Purdue University, January 1993.

[CS93b] D. E. Comer and D. L. Stevens. *Internetworking with TCP/IP*, volume III, Client–Server Programming and Applications. Prentice Hall, first edition, 1993. ISBN 0-13-474222-2.

[DELO89] W. Du, A. K. Elmagarmid, Y. Leu, and S. D. Ostermann. Effects of Local Autonomy on Global Concurrency Control in Heterogeneous Distributed Database Systems. In *Second International Conference on Data and Knowledge Systems for Manufacturing and Engineering*, pages 113–120. IEEE, 1989.

[DO91] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the sprite implementation. *Software–Practice and Experience*, 21(8):757–785, August 1991.

[Dre90] A. Drexl. Job-Prozessor-Scheduling für heterogene Computernetzwerke (Job-Processor Scheduling for Heterogeneous Computer Networks). *Wirtschaftsinformatik*, 31(4):345–351, August 1990.

[DRS89] B. F. Dubach, R. M. Rutherford, and C. M. Shub. Process-Originated Migration in a Heterogeneous Environment. In *Proceedings of the Computer Science Conference*. ACM, 1989.

[DZ83] J. D. Day and H. Zimmermann. The OSI Reference Model. *Proceedings of the IEEE*, 71(12):1334–1340, December 1983.

[EI87] R. B. Essick IV. *The Cross-Architecture Procedure Call*. PhD thesis, University of Illinois at Urbana-Champaign, 1987. Report No. UIUCDCS-R-87-1340.

[ELZ85] D. L. Eager, E. D. Lazowska, and J. Zahorjan. A Comparison of Receiver-Initiated and Sender-Initiated Dynamic Load Sharing. Technical Report 85-04-01, University of Washington, Department of Computer Science, April 1985.

[Ens78] P. H. Enslow, Jr. What Is a "Distributed" Data Processing System? *IEEE Computer*, 11(1):13–21, January 1978.

[EV87] F. Eliassen and J. Veijalainen. Language Support for Multidatabase Transactions in a Cooperative, Autonomous Environment. In *TENCON '87*, pages 277–281, Seoul, 1987. IEEE Regional Conference.

[FA89] R. Finkel and Y. Artsy. The Process Migration Mechanism of Charlotte. *IEEE Computer Society Technical Committee on Operating Systems Newsletter*, 3(1):11–14, 1989.

[Fin88] R. A. Finkel. *An Operating Systems Vade Mecum*. Prentice Hall, Englewood Cliffs, NJ, second edition, 1988. ISBN 0-13-637950-8.

[FR90] D. G. Feitelson and L. Rudolph. Distributed Hierarchical Control for Parallel Processing. *IEEE Computer*, 23(5):65–77, May 1990.

[Fre] Freedman Sharp and Associates Inc., Calgary, Alberta, Canada. *Load Balancer v3.3: Automatic Job Queuing and Load Distribution over Heterogeneous UNIX Networks*.

[fS87a] International Organization for Standardization. Information Processing Systems — Open Systems Interconnection — Specification of Basic Specification of Abstract Syntax Notation One (ASN.1). International Standard number 8824, ISO, May 1987.

[fS87b] International Organization for Standardization. Information Processing Systems — Open Systems Interconnection — Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1). International Standard number 8825, ISO, May 1987.

[GA90] A. Ghafoor and I. Ahmad. An Efficient Model of Dynamic Task Scheduling for Distributed Systems. In *Computer Software and Applications Conference*, pages 442–447. IEEE, 1990.

[GLR84] C. Gao, J. W. S. Liu, and M. Railey. Load balancing algorithms in homogeneous distributed systems. In *Proceedings of the International Conference on Parallel Processing*, pages 302–306. IEEE, August 1984.

[GMK88] H. Garcia-Molina and B. Kogan. Node Autonomy in Distributed Systems. In *ACM International Symposium on Databases in Parallel and Distributed Systems*, pages 158–166, Austin, TX, December 1988.

[GS91] S. Garfinkel and E. Spafford. *Practical UNIX Security.* O'Reilly and Associates, 1991. ISBN 0-937175-72-2.

[GSS89] C. A. Gantz, R. D. Silverman, and S. J. Stuart. A Distributed Batching System for Parallel Processing. *Software–Practice and Experience*, 1989.

[HB84] K. Hwang and F. Briggs. *Multiprocessor Systems Architectures.* McGraw-Hill, New York, 1984.

[HCG+82] K. Hwang, W. J. Croft, G. H. Goble, B. W. Wah, F. A. Briggs, W. R. Simmons, and C. L. Coates. A UNIX-based local computer network with load balancing. *Computer*, 15(4):55–65, April 1982.

[Hed88] C. Hedrick. Routing information protocol. RFC 1058, Network Information Center, June 1988.

[HPS71] P. Hoel, S. Port, and C. Stone. *Introduction to Probability Theory.* Series in Statistics. Houghton Mifflin Company, 1971. ISBN 0-395-04636-x.

[HS88] D. Hochbaum and D. Shmoys. A Polynomial Approximation Scheme for Scheduling on Uniform Processors: Using the Dual Approximation Approach. *SIAM Journal of Computing*, 17(3):539–551, June 1988.

[HWK89] C. C. Hsu, S. D. Wang, and T. S. Kuo. Minimization of task turnaround time for distributed systems. In *Proceedings of the 13th Annual International Computer Software and Applications Conference*, 1989.

[JLHB88] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[KMS93] A. H. Karp, K. Miura, and H. Simon. 1992 Gordon Bell Prize Winners. *IEEE Computer*, 26(1):77–82, January 1993.

[Knu73] D. E. Knuth. *The Art of Computer Programming, Volume III: Searching and Sorting*. Addison-Wesley, 1973. ISBN 0-201-03803-X.

[KP84] D. Klappholz and H. C. Park. Parallelized process scheduling for a tightly-coupled MIMD machine. In *Proceedings of the International Conference on Parallel Processing*, pages 315–321. IEEE, August 1984.

[KR90] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1990.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[Lit87] M. J. Litzkow. Remote UNIX: Turning Idle Workstations Into Cycle Servers. In *USENIX Summer Conference*, pages 381–384, 2560 Ninth Street, Suite 215, Berkeley, CA 94710, 1987. USENIX Association.

[Lo88] V. M. Lo. Heuristic Algorithms for Task Assignment in Distributed Systems. *IEEE Transactions on Computers*, 37(11):1384–1397, November 1988.

[Lot84] M. Lottor. Simple file transfer protocol. RFC 913, Network Information Center, September 1984.

[Mac93] S. Macrakis. The structure of ANDF: Principles and examples. Technical report, Open Software Foundation, 1993.

[Mal93] G. Malkin. RIP version 2–carrying additional information. RFC 1388, Network Information Center, January 1993.

[MG80] S. Majumdar and M. L. Green. A distributed real time resource manager. In *Proceedings of the IEEE Symposium on Distributed Data Acquisition, Computing, and Control*, pages 185–193, 1980.

[MS70] R. A. Meyer and L. H. Seawright. A Virtual Machine Time-Sharing System. *IBM Systems Journal*, 9(3):199–218, 1970.

[NA81] L. M. Ni and K. Abani. Nonpreemptive load balancing in a class of local area networks. In *Proceedings of the Computer Networking Symposium*, pages 113–118. IEEE, December 1981.

[Nic87] D. A. Nichols. Using Idle Workstations in a Shared Computing Environment. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 5–12. ACM, 1987.

[Nil80] N. J. Nilson. *Principles of Artificial Intelligence.* Tioga Publishing Company, 1980.

[OCD+88] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. The Sprite Network Operating System. *IEEE Computer*, pages 23–36, February 1988.

[OSS80] J. K. Ousterhout, D. A. Scelza, and Pradeep S. Sindhu. Medusa: An experiment in distributed operating system structure. *Communications of the ACM*, 23(2):92–105, February 1980.

[Ost93] S. D. Ostermann. Reliable Messaging Protocols. Ph.D. Dissertation, Purdue University, 1993.

[Ous93] J. K. Ousterhout. *An Introduction to Tcl and Tk.* Addison-Wesley, 1993.

[Pee92] Dr. N. E. Peeling. TDF specification, issue 2.0 revision 1. Technical report, Defense Research Agency, Worcestershire, United Kingdom, December 1992.

[Pet88] L. Peterson. The Profile Naming Service. *ACM Transactions on Computer Systems*, 6(4):341–364, November 1988.

[PH90] C. Partridge and R. Hinden. Version 2 of the Reliable Data Protocol (RDP). RFC 1151, Network Information Center, April 1990.

[PM83] M. L. Powell and B. P. Miller. Process migration in DEMOS/MP. In *Proceedings of the 9th Symposium on Operating Systems Principles (Operating Systems Review)*, pages 110–119. ACM SIGOPS, October 1983.

[Pos80a] J. B. Postel. User Datagram Protocol. RFC 768, Network Information Center, August 1980.

[Pos80b] J.B. Postel. DoD standard Transmission Control Protocol. RFC 761, Network Information Center, January 1980.

[PR85] J. B. Postel and J. K. Reynolds. File transfer protocol. RFC 959, Network Information Center, October 1985.

[PS90] C. C. Price and M. A. Salama. Scheduling of Precedence-Constrained Tasks on Multiprocessors. *Computer Journal*, 33(3):219–229, June 1990.

[Puc88] M. F. Pucci. Design Considerations for Process Migration and Automatic Load Balancing. Technical report, Bell Communications Research, 1988.

[PW85] G. Popek and B. Walker, editors. *The Locus Distributed System Architecture.* The MIT Press, 1985.

[RCD91] S. Ramakrishnan, I. H. Cho, and L. Dunning. A Close Look at Task Assignment in Distributed Systems. In *INFOCOM '91*, pages 806–812, Miami, FL, April 1991. IEEE.

[RS82] J. Reif and P. Spirakis. Real time resrouce allocation in distributed systems. In *Proceedings of the Symposium on Principles of Distributed Computing*, pages 84–94. ACM, August 1982.

[RS84] K. Ramamritham and J. A. Stankovic. Dynamic task scheduling in distributed hard real-time systems. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 96–107. IEEE, May 1984.

[Sar89] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. The MIT Press, 1989.

[SH86a] V. Sarkar and J. Hennessy. Partitioning Parallel Programs for Macro-Dataflow. In *ACM Conference on Lisp and Functional Programming*, pages 202–211, August 1986.

[SH86b] Vivek Sarkar and John Hennessy. Compile-time Partitioning and Scheduling of Parallel Programs. *SIGPLAN Notices*, 21(7):17–26, July 1986.

[Shu90] C. M. Shub. Native Code Process-Originated Migration in a Heterogeneous Environment. In *Proceedings of the Computer Science Conference*. ACM, 1990.

[SM79] L. H. Seawright and R. A. MacKinnon. VM/370—A Study of Multiplicity and Usefulness. *IBM Systems Journal*, 18(1):4–17, 1979.

[Sol92] K. Sollins. The TFTP protocol (revision 2). RFC 1350, Network Information Center, July 1992.

[Spa86] E. H. Spafford. *Kernel Structures for a Distributed Operating System*. PhD thesis, Georgia Institute of Technology, 1986.

[SS84] J. A. Stankovic and I. S. Sidhu. An adaptive bidding algorithm for processes, clusters and distributed groups. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 49–59. IEEE, May 1984.

[Sta81] J. A. Stankovic. The analysis of a decentralized control algorithm for job scheduling utilizing bayesian decision theory. In *Proceedings of the International Conference on Parallel Processing*, pages 333–340. IEEE, 1981.

[Sta84] J. A. Stankovic. Simulations of three adaptive, decentralized controlled, job scheduling algorithms. *Computer Networks*, 8(3):199—217, June 1984.

[Sta85a] J. A. Stankovic. An application of bayesian decision theory to decentralized control of job scheduling. *IEEE Transactions on Computers*, C-34(2):117–130, February 1985.

[Sta85b] J. A. Stankovic. Stability and distributed scheduling algorithms. In *Proceedings of the 1985 ACM Computer Science Conference*, pages 47–57. ACM, March 1985.

[Staly] Standard Performance Evaluation Corporation. *The SPEC Newsletter*, published quarterly.

[Sto77] H. S. Stone. Multiprocessor Scheduling with the Aid of Network Flow Algorithms. *IEEE Transactions on Software Engineering*, SE-3(1):85–93, January 1977.

[Sto93] H. S. Stone. *High-Performance Computer Architecture*. Addison-Wesley, third edition, 1993. ISBN 0201526883.

[Stu88] M. Stumm. The Design and Implementation of a Decentralized Scheduling Facility for a Workstation Cluster. In *Proceedings of the 2nd IEEE Conference on Computer Workstations*, pages 12–22. IEEE, March 1988.

[Sun87] XDR: External Data Representation Standard. Sun Microsystems Inc., June 1987. RFC 1014.

[TLC85] M. M. Theimer, K. A. Lantz, and D. R. Cheriton. Preemptable Remote Execution Facilities for the V-System. In *Proceedings of the Tenth Symposium on Operating Systems Principles*, pages 2–12, December 1985.

[VHS84] D. Velten, R. Hinden, and J. Sax. Reliable Data Protocol. RFC 908, Network Information Center, July 1984.

[vR92] G. van Rossum. *Python Reference Manual*. Dept. CST, CWI, 1098 SJ Amsterdam, The Netherlands, April 1992.

[VW84] A. M. Van Tilborg and L. D. Wittie. Wave scheduling—decentralized scheduling of task forces in multicomputers. *IEEE Transactions on Computers*, C-33(9):835–844, September 1984.

[WM85] Y. T. Wang and R. J. T. Morris. Load sharing in distributed systems. *IEEE Transactions on Computers*, C-34(3):204–217, March 1985.

[WS90] L. Wall and R. L. Schwarz. *Programming* **perl**. O'Reilly & Associates, 1990. ISBN 0-937175-64-1.

[WV80] L. D. Wittie and A. M. Van Tilborg. MICROS, a distributed operating system for MICRONET, a reconfigurable network computer. *IEEE Transactions on Computers*, C-29(12):1133–1144, December 1980.

[WW90] C. M. Wang and S. D. Wang. Structured Partitioning of Concurrent Programs for Execution on Multiprocessors. *Parallel Computing*, 16:41–57, 1990.

[Zay87] E. Zayas. Attacking the process migration bottleneck. In *Proceedings of the 11th Symposium on Operating Systems Principles*, pages 13–24. ACM, November 1987.

VITA

VITA

Stephen Joel Chapin was born in Oberlin, Ohio, USA on July 9, 1963. In May, 1985 he graduated *cum laude* from Heidelberg College in Tiffin, Ohio with a Bachelor of Science degree in Computer Science and Mathematics. He entered Purdue University in August of 1985, and received a Master of Science in Computer Science from Purdue University in May of 1988. He began his doctoral work under Professor Eugene H. Spafford in 1991. His research interests include operating systems, distributed systems, heterogeneous computing systems, networking protocols, and parallel computing.