An Overview of the MESSIAHS Distributed Scheduling Support System*

Technical Report TR-CSD-93-011

Steve J. Chapin Eugene Spafford
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907-1398
sjc@cs.purdue.edu spaf@cs.purdue.edu

January 22, 1993

Abstract

Users often find that local resources are too limited to solve large computing problems. At the same time, unused machines remain inaccessible because of incompatible architectures, ignorance of their capabilities, or incompatible administrative restrictions. To preserve this investment in equipment, yet allow for the solution of large problems, mechanisms are needed to join these systems into cooperating groups across the boundaries of administrative domains and physical locality.

In this paper, we describe MESSIAHS, a system intended to provide scalable mechanisms for the efficient implementation of scheduling policies on distributed systems, while preserving the autonomy of the component systems. These systems can range from a few workstations to hundreds of heterogeneous, autonomous systems interconnected via networks ranging from local-area networks to geographically large networks, connected by arbitrary links.

*This work was supported by a NASA Graduate Student Researchers Fellowship, NASA grant number NGT 50919.

1 Introduction

In a typical research environment, there is a large investment in computing equipment, typically dozens or hundreds of workstations. Studies have shown that this equipment is usually underutilized (see Gantz, et al. [18] and Litzkow [26]). Users of this equipment often find that their local resources are not sufficient to execute large programs, while the combined resources of several machines might solve the problem at hand.

A solution to this problem is to conglomerate the separate processors into a distributed system, and then to join the distributed systems into larger systems to further expand the computational power of the whole. Until now, obstacles such as incompatible architectures and restrictive administrative domains have blocked the formation of large-scale distributed systems that are composed of heterogeneous, autonomous systems.

The MESSIAHS[1,2] project is investigating mechanisms to connect and support task placement in autonomous, heterogeneous, distributed systems. These mechanisms might be used by a scientist who wishes her job to be run on a large system, but is unconcerned whether the computer is located in the same building or across the country. The job may have subtasks best suited for a large vector processor or a tightly-couple parallel machine, and the output might be rendered on a graphics workstation.

In a conventional situation, the scientist would have to discover which processors are currently available, and then reserve them for computation. This process is inefficient, and decreases the realized throughput on these machines. Using MESSIAHS, she would submit her individual tasks to the scheduling system running on her workstation, and MESSIAHS would automatically locate suitable execution sites and schedule the tasks for execution.

An important distinction must be drawn 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, while 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.

The remaining sections of this paper define terms (§2), describe the elements necessary to solve the problem (§3), the principles guiding the development of our scheduling mechanisms and the complications inherent in scheduling for autonomous systems (§4), our proposed approach to solving the problem of supporting distributed scheduling (§5), an abstract implementation based on events (§6), a formal model for information updates (§7), and our concluding remarks (§8).

[1] **M**echanisms **E**ffecting **S**cheduling **S**upport **I**n **A**utonomous, **H**eterogeneous **S**ystems.

[2] mes.si.ah n: a professed or accepted leader in some hope or cause.

2 Definitions and Motivation

Before we can discuss our solution in detail, we must first define the terms we will use to describe the problem.

Autonomous systems consist of one or more subsystems connected by a communication medium; at the lowest level, a processor is an autonomous system with no subsystems. A key feature of autonomous systems is that all information, behavior, and policy pertaining to a system is private to that system. Any sharing of private information is at the discretion of the local system.

Because of the prevailing decentralization of computing resources, distributed computation systems must support autonomy. There is usually no longer a single, authoritative controlling entity for the computers in a large organization. A scientist may control a few machines of his own, and his department may have administrative control over several such sets of machines. That department may be part of a regional site, which is, in turn, part of a nationwide organization. No single entity, from the scientist to the large organization, has complete control over all the computers it may wish to use.

Heterogeneous systems are multiprocessor systems that may have processors of different types. They may have different architectures, computation speeds, operating systems, and devices. In contrast, homogeneous systems have the same architecture and operating system, although they may vary in performance.

Heterogeneity is important because it yields the most cost-effective and efficient method for performing some computations. For example, a large computation might have pieces best suited for execution on a vector-processing supercomputer, while other parts might run best on a massively parallel machine or a graphics workstation. If the program is restricted to using only one architecture within the distributed system, it will suffer needless delay.

Distributed systems communicate by passing messages over an external communications channel. Such systems are often called multicomputers (as defined by Spafford in [41]) or loosely-coupled systems, as opposed to tightly-coupled parallel machines that communicate through shared memory.

There are many examples of systems that share some of these qualities. Shared-memory parallel processors such as the Sequent Symmetry [38] are homogeneous, tightly-coupled systems. Homogeneous groups of workstations that communicate over a network are described by Babin [2], Gantz, et al. [18], and Litzkow [26], among others. Stumm describes a heterogeneous, distributed system of workstations in [43]. None of these systems support autonomy.

Our definition of a *task* is a request for resources. This includes the conventional model of a computationally intensive unit in a larger program, as well as a set of database queries (see Carey, et al. [7]), output requests, etc. Thus, our mechanisms could be used in the scheduling of queries to a large distributed database, to manage a set of output devices, such as printers, or to allocate network resources for large data transfers. For simplicity of description, we will restrict our discussions to the conventional model of placing computational tasks on processors.

Within a distributed system, there are two levels of task scheduling: the association of tasks with processors, also called task placement, and the choice of which task to execute among those available on a processor. Our work concentrates on facilities for the former.

Many researchers have studied the problem of task placement in distributed systems, including Sarkar [37], Lo [28], Stone [42], and Blake [4]; however, their algorithms have assumed that all processors are similar, that perfect information describing the system and tasks is instantaneously available, or that they have total control of all processors in the system. We have found no evidence in the literature that our more general case, with heterogeneous and autonomous systems, has been studied. We have likewise found no reports describing the mechanisms necessary to implement algorithms for such systems.

The scheduling support mechanisms we are developing will support systems that are autonomous, heterogeneous, and distributed. Unless noted otherwise, all uses of the terms *autonomous system* and *system* in this paper refer to autonomous, heterogeneous systems. An *administrator* is an entity, either a human or a software module, that decides the policy for a system. A *user* submits tasks to a system for processing.

3 Requirements

To build a hierarchical, heterogeneous distributed system, we require several hardware and software components. Most obviously, we require an interconnection network for the processors, and that each processor be able to send and receive messages on the network.

If the processors are to exchange tasks, there must be a mechanism to move programs, and autonomy demands an associated mechanism to revoke or migrate a running task. For example, a computation might start on an unused workstation at night. If the local scheduling policy determines that the job should no longer be run, the acceptance of the program must be revoked, and the job would have to be migrated or restarted within the system. This might occur if the user returned to his workstation in the morning, or if the load average rose above a threshold. Without a revocation facility, autonomy is not possible, and the available processing power may be decreased as users refuse to allow their machines to run jobs from the distributed system, rather than sacrifice their sovereignty. Note that checkpoints would normally be employed in such a situation to prevent the loss of partial results.

Message passing protocols, both reliable and unreliable, are necessary to exchange system description and task description information. The content of these messages must be represented in an architecture-independent format. The information in the descriptions must be descriptive enough to be used by existing scheduling algorithms, and the mechanism for expressing it must be extensible to accommodate new algorithms.

To be practical, the system must provide a mechanism for the administrator to express a scheduling policy. It must also allow the administrator to control the flow of information out of the system to support autonomy and security.

3.1 Prior work

Several of these problems have been investigated by other researchers. The FTP protocol family (see [35, 40, 29, 11]) provides mechanisms to move files and programs. Efficient implementations of revocation might use checkpointing and process migration mechanisms, such as those as in Emerald [24], Amoeba [30], Sprite [31], Dune [36], Charlotte [1], or the V System [44]. These mechanisms can also be used in developing fault tolerance and load balancing schemes.

Essick [15], and Shub, et al. [13, 39]) have devised architecture-independent task representations. XDR [23] and ASN.1 [16, 17] specify machine-independent data formats. The User Datagram Protocol [34] and Reliable Datagram Protocol [45, 32] protocols from the DARPA TCP/IP protocol suite are message-based protocols, and could form the basis for our information exchange prototols.

3.2 New research

The remaining components constitute our research. We have designed protocols to exchange information vectors describing the resources available in a system and describing the tasks that are candidates for execution. The information contained in the descriptions consists of a fixed portion and a general-purpose extension mechanism to facilitate the implementation of new scheduling algorithms.

We have designed and constructed software modules to interpret a description of a scheduling policy and enforce that policy in decision making. A similar interpretive mechanism mediates the combination of multiple system descriptions into one, for further propagation.

The autonomy, distribution, and heterogeneity constraints complicated the design and construction of each of these components. The difficulties posed by these factors are discussed in depth in section 4.

3.3 Assumptions

In addition to supplying the required components, we make several assumptions about machines that participate in our autonomous system:

- Any machine that submits tasks to the system reciprocates by accepting scheduling requests, but not necessarily honoring them. This promotes "fair play," and that a greedy user could not take advantage of others' generosity while hoarding his own machine's capabilities for himself.

- The makeup of the system is dynamic; machines are free to leave or join the system at any time. Machines may crash, or they may become unavailable for task scheduling because of policy considerations.

- Communication between systems is not reliable. Networks often experience brief periods of arbitrary delay and data loss.

- A subsystem may report to more than one parent system. In an environment where different researchers from different administrative domains pool their funds to purchase a large machine, both domains may wish to schedule tasks upon the machine.

4 Guidelines and Constraints

We have five guiding principles for our scheduling mechanisms:

1. Strive for generality. Because we cannot foresee all requirements of scheduling algorithms, the mechanism must be extensible. In particular, the representations of systems and tasks must adapt to the requirements of users.

2. Preserve local autonomy. There should be no forfeiture of local control. The mechanisms must support the autonomy of the policy for each system; only those data the local policy wishes to advertise should be advertised. Each machine within the system is free to have a local scheduling policy that does not conform to a global policy, and the mechanisms must support this. Configurability is important to maintain autonomy. Parameters that control the system's behavior should be tunable, when possible.

3. Facilitate scalability. The architecture should function on systems ranging from a single workstation to hundreds or even thousands of processors, with interconnection schemes ranging from local area networks to wide area networks. Centralization of information must be avoided to achieve this ability to scale.

4. Minimize overhead. The monitoring overhead and message traffic must be kept low so as to not adversely impact performance within the system, i.e. the scheduling mechanism must minimize its interference with the running of application programs. The scheduling module should also minimize use of memory and disk resources.

5. Maintain data integrity. The mechanisms must support scheduling algorithms by providing accurate information. An optimal support mechanism would have

all the information required by an algorithm available at all times, and this information would be perfectly accurate. We are concerned with the timeliness, the completeness, and the accuracy of the data available to the algorithms.

As we examine the complications created by the interaction of these guidelines and the three basic factors of heterogeneity, autonomy, and distribution, we will explain how these principles influenced our decision making during the design process.

4.1 Conflicts between guidelines

Note that although these guidelines can be adhered to individually, they have mutually exclusive tenets when taken collectively. We now examine the conflicts that arise when two or more of the principles are observed simultaneously. Solutions to these compromises are detailed in sections 5 and 6.

Overhead and data integrity

Task placement algorithms require descriptions of the tasks to be scheduled as well as of the distributed system. The quality of the mapping of tasks to processors can depend upon the accuracy of the information available to the algorithm. The more timely and accurate the description information is, the better.

The description of the distributed system is highly dynamic. Because the system is made up of many separate computers, there is a large amount of information to be gathered. Typical system description information includes processor speed and utilization, available memory, and communications delay between systems. Because of factors such as processor and network load, and the freedom of machines to join and leave the system at will, this information can change rapidly.

If the scheduling algorithms are to have accurate information describing this dynamic system, the mechanisms that gather this information must allow for frequent updates. This is at loggerheads with the principle of minimal overhead.

Spatial overhead can be as important as processor or network overhead. Data representations with higher precision can require additional storage, and thus increase overhead.

Scalability and data integrity

Scalability precludes the centralization of information. If the system is to be accurately represented, then complete information must still reach all nodes. The decentralization of data storage introduces latency which degrades the timeliness of the data.

Generality and overhead

To support generality, the system must have an extension mechanism that allows the system and task representations to be augmented. Use of these extensions increases the overhead for processing a description.

Overhead and scalability

The replicated, distributed data storage required by the scalability guideline consumes more storage space than its centralized counterpart.

4.2 Autonomy considerations

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 central authority.

In prior work, Garcia-Molina and Kogan [19], and Eliassen and Veijalainen [14] examined autonomy in distributed systems and devised taxonomies for its different types. The Eliassen and Veijalainen scheme is more general than Garcia-Molina, but is not as detailed. We combined the two schemes and tailored the category descriptions to our application of distributed scheduling.

We define four classes of autonomy:

design autonomy (D-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. This gives rise to heterogeneity, as machines can have distinct instruction sets, byte orderings, processor speeds, operating systems, etc.

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

execution autonomy (E-autonomy) Each system decides whether it will honor a request to execute a task, and has the right to revoke a task it had previously accepted. The local policy decides what resources are to be shared.

administrative autonomy (A-autonomy) Each system can set its own usage policies, independent of others. 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 system can attempt to join any other system; conversely, the module managing the administrative aspects of a system can refuse any such attempt to join.

Because of E-autonomy and C-autonomy, all decisions pertaining to a system are under its control. It advertises as little or as much of its system state as its local policy decrees, and cannot be forced to accept tasks for execution. Because of this, we cannot be sure that we have complete information describing a system; we only know what it chooses to tell us about itself. This can compromise data accuracy if a pertinent statistic is not advertised.

The E-autonomy constraint requires our 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.

The combination of E-autonomy and heterogeneity poses another problem for migration. We must move a process from one machine to another, but because of C-autonomy, we may not know the architecture of the recipient machine. Therefore, advance translation of the program image might be impossible. Essick [15] and Shub, et al. [13, 39] have examined the problem of compiling programs for multiple architectures, but their solutions are of limited applicability to the problem. An ideal solution would be an analog of XDR for programs. For the moment, processors with different instruction sets cannot directly share code, and process migration between heterogeneous systems is not supported.

A-autonomy means that we cannot rely on neighboring systems to behave in any specific manner. When combined with C-autonomy, it means that expected inter-message times may be quite different between neighbors. Combining A-autonomy and E-autonomy means that the local scheduling policy might act to defeat the concerted efforts of a group of cooperating remote modules.

A small amount of C-autonomy is lost when the system conforms to the generality guidelines, because the format of its transmitted data is defined externally.

4.3 Distribution considerations

The distributed and autonomous nature of the system preclude global sharing of information. Lamport, in [25], showed that information in a distributed system will always be out of date. Thus, there will always be some latency involved in the information reporting. We cannot know what the system looks like instantaneously; we must content ourselves with an estimate of what it looked like at some point in the past. There is an obvious tradeoff between freshness of data and minimization of resources spent maintaining the data. MESSIAHS provides facilities to tune its behavior, giving the administrator freedom to set the balance.

A side effect of scalability and distribution is that we cannot keep complete information on every processor in the system. The bookkeeping requirements would quickly consume the processing power of the system, and little or no productive work

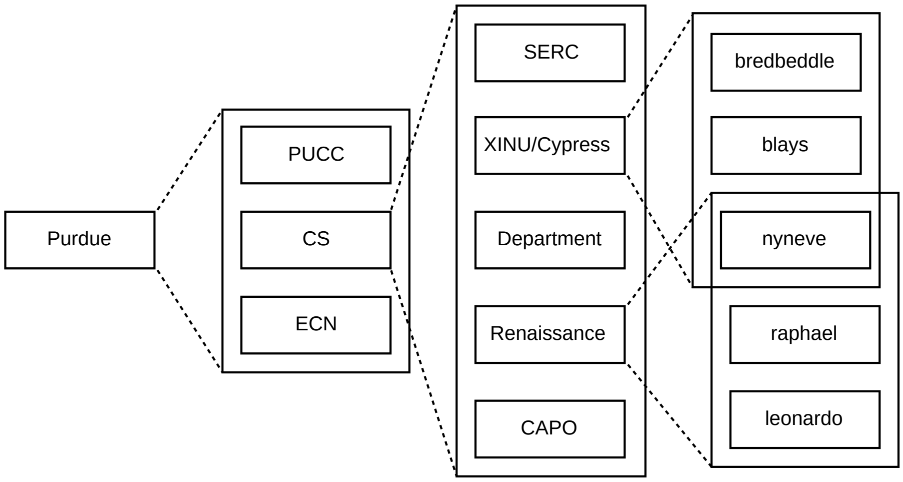

Figure 1: A Sample Autonomous System Architecture

would be accomplished. Again, we have a tradeoff between the accuracy of a system description and its size. We have designed a mechanism to combine and compress multiple system descriptions into one, thus saving space while preserving much of the descriptive information.

4.4 Heterogeneity considerations

Heterogeneity is a special case of D-autonomy, but is significant enough to deserve special mention. Because the individual processors within the system can have different architectures (different data representations, machine instructions, execution speed, etc.), values passed between machines must use a standard external data representation, such as XDR [23]. A program compiled for one architecture cannot be directly executed on a machine of a different architecture.

5 The Architectural Model

We intend to build our autonomous systems in a hierarchical fashion, which is a model often used in the real world. An autonomous system is composed of a set of subordinate autonomous systems. Within each of these sets there can be many machines, which could be further grouped into autonomous systems. At the lowest level, each machine is the sole member of an autonomous system. For example, the set of computers at Purdue University is an autonomous system. Within the Purdue hierarchy, there are many subordinate autonomous systems, including those used by the School of Engineering (ecn), the Department of Computer Sciences (cs), and the Computing Center (pucc). The computer science machines decompose into several groups: the general-use department machines, the Software Engineering Research

Center, the XINU/Cypress project, and the Renaissance project, among others. This is pictorially represented in figure 1.

Networks are *not* autonomous systems; sets of machines are. Autonomous systems are logical, administrative groupings; they may or may not correspond to physical groupings of machines. The interconnection network for a set of machines may suggest an efficient grouping of autonomous systems. 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 autonomous system. The nyneve node is an example of a machine under administrative control of two research projects, the XINU project and the Renaissance project, and therefore belongs to two autonomous systems.

We represent our systems with acyclic, directed graphs, with encapsulating autonomous systems as interior nodes and machines as leaf nodes. Each node is the root of an autonomous system, and is virtual. In some cases, the virtual nodes map directly onto machines, and in other cases they do not; e.g. there is a machine leonardo at Purdue, but there is not necessarily any machine named CS or Renaissance. A machine within an autonomous system is selected to act as its representative to higher levels within the graph. So, blays might act as the XINU/Cypress spokesman within the CS system.

We have defined autonomous systems as hierarchical constructs, where an autonomous system is made up of one or more subordinate systems. We call an encapsulating autonomous system a *parent*, and a subordinate system a *child*. Thus, in our 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*. The term *neighbor* refers to one of a node's parents, children, and siblings.

In MESSIAHS, each autonomous system in the hierarchy has a scheduling support module that is responsible for maintaining the set of information required by the scheduling policy and for moving tasks between systems.[3] It provides the mechanism upon which the scheduling policy is built. There are two facets to the local policy that our modules support: task placement and task acceptance. Task placement algorithms take a set of tasks and a description of the underlying multicomputer and devise an assignment of tasks to processors according to an optimizing criterion. Because our systems are autonomous, each has a local policy to determine if a task assignment will be accepted.

Our method for implementing the module has three main parts: the *system description vector* (SDV), the *task description vector* (TDV), and the protocol used to communicate between systems.

[3]We will often use the notation "X" as a shorthand for "the scheduling module for autonomous system X."

5.1 The system description vector

The system description vector encapsulates the state of a system and is used to advertise its abilities to other systems that may request it to schedule tasks. The vector is the information base a scheduling module uses to choose a candidate system for a task from among its neighboring systems.

The system description vector is designed to support the scheduling of conventional tasks. The flexibility of our mechanism allows us to tailor the vector to other applications.

To determine what information should be passed in the vector, we surveyed the existing research and noted the classes of information used for scheduling algorithms (including Sarkar [37], Lo [27, 28], Stone [42], Drexl [12], and Hochbaum and Shmoys [22]). Surprisingly, very few of the current scheduling algorithms use any information beyond processor speed. Of the surveyed papers, 50% used the processor speed as input to their algorithms, while only 7% considered the communications structure of the system. The relationship between the SDV and TDV are shown in table 1.

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

Table 1: Comparison of TDV to SDV

This is a classic “chicken and the egg” problem: which comes first, the demand for specific system information by complicated algorithms, or the information that makes such algorithms feasible? We posited that if such information were readily available, more algorithm designers would use it, and we augmented the set with items we expect will be desired in the future. Factors in the resulting set include:

- memory statistics (available and total)
- processor load (queue length, average wait time, and processor utilization ρ)
- system characteristics (processor speeds, the number of processors, etc.)
- communication costs (point-to-point, start up, read/write)

- a measure of the system's willingness to take on new tasks

Each scheduling support module within the system has the ability to cache information to build a history of behavior for subordinate systems. This history mechanism is user-configurable, and information on the history characteristics of each datum (e.g. permanence, mean, standard deviation, etc.) is passed with the datum.

Our design defines a static set of machine classes for each characteristic. The classes are based on a logarithmic scale. We chose the logarithmic scale because of its applicability to real-world processors. This embodies the balance between scalability and minimization of overhead: we predefine a fixed set of machine classes, yielding efficient processing, while condensing the information to achieve data compaction and the ability to expand to large systems.

If a system provides special services, such as specialized I/O devices, vector processors, etc., it can use the extension mechanism described in section 5.4.

5.2 The task description vector

The task description vector is analogous to the system description vector; it represents the resource requirements of a task. The task vector is used by the task acceptance facet of the scheduling policy in conjunction with system description. 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. 64% of the algorithms computed results based on the estimated run-time of a task, and 57% used inter-task communication estimates. Our task description vector consists of:

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

Tasks that require special services will describe them using the same extension mechanism used for the system description vectors.

5.3 The protocols

The communications protocols define the interaction between scheduling modules within the autonomous 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.

5.3.1 *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 application-defined data. The interpretation of the application-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 [5], 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.

At periodic intervals a module recomputes its status vector and advertise the updated vector through the update channel. The length of the period is a locally tunable parameter (recall the discussion in section 4.1 on the tradeoff between overhead and data integrity. A short timeout period ensures that update recipients have an accurate view of the sender, but incurs a penalty in terms of machine load. A long timeout is computationally cheap, but risks the development of inaccurate schedules based on outdated information. When the countdown timer for the period expires, the scheduling support module recomputes the state representation for its autonomous system, and advertises it. This is done regardless of how recently it received updates from other systems. We will investigate the use of a multicast facility for this channel, such as described by Birman, et al. in the ISIS system manual [3], and by Deering in [10]. Provision is also made for polled updates, whereby a system can query another as to its status through the control channel and receive a reply through the update channel.

Update cycles cannot be allowed in the communications structure of a system. An update cycle occurs when two or more systems exchange update messages and compute their status vectors based on those messages. Such behavior causes an ever-increasing overestimation of system resources, analogous to the *count to infinity* problem in network routing protocols (see Comer [9]). For any system, there are three sets of systems that could pass it updates: its children, its parents, and its siblings within the hierarchy. In order to avoid update cycles, we do not allow parents to pass update messages to their children. Also, we tag the updates passing from child to parent; these are the only updates used in the computation of the system description vector. Updates from siblings are passed to the task placement module, but are not incorporated into the system vector.

5.3.2 *The control channel*

The control channel is intended to be a bidirectional, reliable, message-based channel, such as described by Hinden, et al., in the Reliable Datagram Protocol description [32, 45]. 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 defined control message types are **sched_request**, **sched_accept**, **sched_deny**, **task_request**, **task_accept**, **task_status_query**, **task_status**, **task_kill**, and **system_status_query**.

sched_request

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

sched_accept

> The recipient of a **sched_request** accepts the request by replying to the requester with this message. The data for this message includes the identification number of the accepted **sched_request** message.

sched_deny

> The recipient of a **sched_request** message passes its refusal to accept the request to the requester. The data includes the identification number of the rejected **sched_request** message.

task_request

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

task_accept

> The system accepts the task request. The data for this message includes the identification number of the accepted **task_request** message. The task is moved through the task channel.

task_deny

> 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.

task_status_query

> The sender submitted a job for execution, and is requesting information on the status of the task. The **task_request** and job identifier of the task are included

in the message. This message is useful to test for abnormal conditions such as a link failure in the network, or a system crash.

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 six possible states: executing, finished, aborted, killed, revoked, and denied.

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 an error has occured, e.g. a bus error or division by zero. The *finished* state does not guarantee the correct functionality of the task, only that it did not crash.

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 *revoked* message is sent. A *denied* reply means that the requested system would not accept the task for execution.

task_kill

The sender requests that the receiver stop executing the task named in the message. If the receiver chooses to do so, it should return a **task_status** message.

system_status_query

Query the state of a system. A system description vector will be returned through the update channel in response to this request.

5.3.3 *The task channel*

The task channel reliably transfers a task between two nodes in an autonomous system. Once a task's destination has been negotiated using the control channel, a task channel is opened 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 an autonomous system that prohibits an outside system from directly accessing its members. In this case, the task is delivered to the encapsulating autonomous system, which is then responsible for forwarding the task to its destination. Garfinkel and Spafford [20] define this type of behavior as a *firewall*. Cheswick discusses the the construction of a secure packet router embodying the firewall concept in [8].

5.4 The extension mechanism

It would be impossible to predefine the complete set of characteristics used by all present and future scheduling algorithms. Therefore, MESSIAHS includes an extension mechanism that allows users to customize the description of a system or task. Users may append a set of `(variable, value)` pairs to the description vector, and may specify a predicate in terms of basic functions and these variables. The scheduling module evaluates the predicate, and makes a decision based on its boolean-valued result.

Our extension mechanism is similar in concept to the attribute-based descriptions of the Profile [33] and Univers [6] systems developed at the University of Arizona. We have simplified and tailored the concept to our more limited purposes.

The MESSIAHS extension mechanism has two basic variable types: strings and integers. The extension language includes parentheses for grouping, the comparators `< > = <= >= <>` for integers and `eq gr ls le ge ne` for strings, and the logical operators `and or not xor`. Within the predicate, variable names begin with the character `$`. For example, if the user defined a value of `(cpu, sparc)` for a system, then a predicate might be `($avail_mem >= 4000000) and ($cpu eq "sparc")`.[4] This syntax is similar to that used in the **perl** language [46].

The mechanism is the same for extending both task and system description vectors, but the interpretation of the fields is different. The fields describe the requirements of a task or the capabilities of a system. In the case of a **sched_request** message, the module evaluates its local predicate with values from the received task description; for a **task_request** message, it evaluates the received predicate with values from the description vectors for the available tasks. One can think of this process as acting as a *task filter*.

In concrete terms, the task filter takes as input a description vector and a predicate. The filter parses the description, and wherever it finds a variable name (e.g. $cpu), it fills in the value from the description vector. When all variables have been replaced with their values, the filter evaluates the predicate.

A resulting value of `true` indicates that the vector matches the predicate, and `false` indicates failure. The use of an uninitialized field automatically causes the predicate to return `false`. In the case of a **sched_request**, a `true` value means that the system can accept the job for execution. Similarly, the task filter returns a value of `true` for a `task_request` if a migratable task matches the predicate.

6 Event-based semantics

The semantics of a scheduling module are defined in terms of events and consequent actions. Events in MESSIAHS are either *timeout* events or *message* events. Timeout events occur when a timer expires, and message events indicate the receipt of a message through the communications channel.

[4] Recall that available memory is one of the fixed fields in a description vector.

6.1 Timeout events

Timeout events are one of two types, *input* and *output*. Input timeouts indicate that we have not received an update from a neighboring system (either child, parent, or sibling). Output timeouts indicate that it is time to advertise system state to neighboring systems.

When an output timeout event occurs, the scheduling module must recompute its update vectors and advertise them to the appropriate neighbors. Local policy determines the length of the timeout period.

An input timeout event occurs when a module has not heard from a neighboring module within a prescribed time period. Depending on local policy, the module will either declare the neighboring system down, or it will query the system status of the neighbor through the control channel. Because of the dynamic nature of system membership, this is not an unexpected or erroneous condition.

There are provisions for operating the system in a poll-driven manner, so that input and output timeout events are not generated.

6.2 Message events

Each message type corresponds to a message event. Thus, there are 10 message events: **sched_request**, **sched_accept**, **sched_deny**, **task_request**, **task_accept**, **task_deny**, **task_revoke**, **status_query**, **join_request**, and **status_vector**. The following list details the actions of a scheduling module upon receipt of each type of message. The **status_vector** message is sent through the update channel; all others pass through the control channel.

sched_request

> A scheduling request message contains a description of a task that the sender would like the receiver to accept. The TDV is compared to the SDVs for the module itself, its children, and its parents. If the module accepts the task for itself, it replies with a **sched_accept** message. If not, then it forwards the **sched_request** message to the most likely candidate[5]. If the local module is using proxy accept, and the request is being sent to a child, then the module must replace the source ID in the request with its own, and make note of the change for later task forwarding.

sched_accept

> A **sched_accept** event indicates that the task has been accepted for execution, and is sent in response to a **sched_request**. If the corresponding request has a source ID of the local scheduling module, then either proxy accept is in force, or

[5] Our initial implementation prohibits a module from attempting to concurrently schedule a task on multiple neighbors.

the local module originated the request. In the former case, the module replaces the source ID with the ID earlier, and forwards the acceptance. In the latter case, the module opens a task channel to the acceptor and passes the task. If the source ID is not that of the local module, the acceptance is forwarded.

sched_deny

A **sched_deny** message event indicates that the sender will not accept the task. The receiving module will then send a **sched_request** message to the next eligible system, according to the local policy. If there are no more eligible systems, the denial is forwarded back towards the original sender of the **sched_request** message.

task_request

A task request message contains a description of a task that the sender is willing to run. The TDV is compared to the TDVs of any tasks on the local system, according to local policy (there may be certain tasks the local module is unwilling to migrate). If the module has an eligible task, it replies with a **task_accept** message. If not, then it passes a **task_request** message to the most likely candidate among its neighbors. Just as with **sched_request messages**, if the local module is using proxy accept, and the request is being sent to a child, the module must replace the original source ID with its own.

task_accept

This event indicates that the sender has a task that matches the description from the request message. The receiving module compares the source ID of the requester to its own ID; if they match, and it the module is using proxy accept, it replaces the acceptors id with its own and forwards the acceptance. If the local module is the original requester, it opens a task channel to the acceptor and receives the task. If the source IDs do not match, the acceptance is forwarded.

task_deny

The sender indicates that it has no tasks available matching the request. The local module queries the next eligible system for a task; if none remain, it forwards the denial.

task_revoke

The recipient attempts to reschedule the task at another node, in accordance with local policy. If there are no more eligible nodes, the revocation message is forwarded. If no suitable system can be found to execute the revoked task, the task is aborted.

system_status_query

In response to a **status_query** message event, the receiver returns the appropriate SDV through the update channel. Local policy may limit the information sent.

task_status_query

The sender is asking if the task described in the message is still running. The receiver, if it responds, sends a **task_status** message.

task_status

In response to a **task_status_query** message event, the receiver returns the task's status through the update channel.

join_request

If the local configuration and policy allow, the local module will add the sender as a child system.[6]

status vector If the sender is a child, sibling, or parent, the module saves the vector for later use in scheduling, and resets all timers and counters associated with the link.

7 A Formal Model for Update Vectors

Until now, we have assumed that a correct mechanism exists for combining update vectors. In this section, we define an operation for combining update vectors and analyze the semantics of the operation.

In general, a distributed system is a multigraph, with processors represented as nodes and communications links as edges. In this section, we examine the semantics of combining update vectors distributed systems constructed as a forest of trees, and prove that our update protocol does not overestimate system resources in such systems.

7.1 Definitions and notation

When discussing the system description vectors passed through the update channel, we use a set notation. Members of the set are names of systems; the name of a system represents its capabilities in the update vector. For example, the set $\{a, b, c\}$ contains a description vector that represents the capabilities of a, b, and c. The identifying features of the individual machines, such as names and addresses, are not contained in the update vector.

[6] Cycles should be avoided among autonomous systems. At present, it is the responsibility of the administrator to ensure they do not occur.

We define the operator $\uplus$ to be the set union operator, with duplicate inclusion. Our sets can have duplicate items, which are known as *bags* or *multisets* (such as in the programming language ABC [21]). For example,

$$\{a, b\} \uplus \{a, c\} = \{a, a, b, c\}$$

We use the $\uplus$ operator to coalesce our representation of two SDVs into one. We use multisets because there can be duplicate machines within an autonomous system. This situation is commonplace in environments containing laboratories of public workstations. We will use the notation $\biguplus_{i \, \in \, range} S_i$ to map the $\uplus$ operator over multiple sets.

The operation $A \uplus B$ provides an upper bound on the information contained in the SDV representing their combined capabilities. This is an upper bound because, as per the autonomy constraint, a system may discard part of the incoming SDV before forwarding it[7].

In order to reason about the structure of the system graphs, we define four predicates defining the child, parent, ancestor, and descendant relation between two nodes.

$$parent(p, c) = \begin{cases} true & \text{there is an edge from } p \text{ to } c \text{ in the DAG} \\ false & \text{otherwise} \end{cases} \tag{1}$$

$$child(c, p) = \begin{cases} true & \text{there is an edge from } p \text{ to } c \text{ in the DAG} \\ false & \text{otherwise} \end{cases} \tag{2}$$

$$ancestor(a, d) = \begin{cases} true & \text{if } parent(a, d) \text{ or} \\ & (\exists p \ni \{parent(p, d) \text{ and } ancestor(a, p)\}) \\ false & \text{otherwise} \end{cases} \tag{3}$$

$$descendant(d, a) = \begin{cases} true & \text{if } child(d, a) \text{ or} \\ & (\exists c \ni \{child(c, a) \text{ and } descendant(d, c)\} \\ false & \text{otherwise} \end{cases} \tag{4}$$

In addition, we denote the set of parents of a node x as Pa_x, its children as Ch_x, its siblings as Si_x, its ancestors as An_x, and its descendants as De_x. Equations 5 through 9 give formal definitions for these sets in terms of the four predicates.

$$Pa_x = \{p \mid parent(p, x)\} \tag{5}$$

$$Ch_x = \{c \mid child(c, x)\} \tag{6}$$

[7]We assume that no system will be intentionally deceitful by inflating the capabilities it receives from subordinate systems

$$Si_x = \{s | (x \neq s) \text{ and } (\exists p \ni parent(p, x) \text{ and } parent(p, s))\} \tag{7}$$

$$An_x = \{a \mid ancestor(a, x)\} \tag{8}$$

$$De_x = \{d \mid descendant(d, x)\} \tag{9}$$

7.2 Computation of update vectors

Each node within our autonomous system will compute three sets of system status data to be passed through the update channel. The three sets are intended for the node's parents, children, and siblings, if any. We denote these sets, for a node x, as U_x, D_x, and S_x respectively. These quantities are recursively defined in terms of the structure of the system.

$$U_x = \{x\} \uplus \left(\biguplus_{\forall i \in Ch_x} U_i \right) \tag{10}$$

$$D_x = \{x\} \uplus \left(\biguplus_{\forall j \in Pa_x} D_j \right) \uplus \left(\biguplus_{\forall k \in Si_x} S_k \right) \tag{11}$$

$$S_x = U_x \tag{12}$$

The update vector passed to parents and children is different than the internal data kept for decision making. At any moment, a node will have an internal representation of the data from each child and parent. For two nodes x and y, we define a *view vector* V_{xy}, read as "x's view of y:"

$$V_{xy} = \begin{cases} U_y & y \in Ch_x \\ D_y & y \in Pa_x \\ S_y & y \in Si_x \\ \{x\} & x = y \\ \emptyset & \text{otherwise} \end{cases} \tag{13}$$

At any time, a node x will have $\|Ch_x\| + \|Pa_x\| + \|Si_x\| + 1$ different views to compare with a submitted job.

Our examples are based on the autonomous system depicted in figure 2. The solid lines indicate parent/child links. The dashed lines also indicate parent/child links, but under certain circumstances are treated as missing.

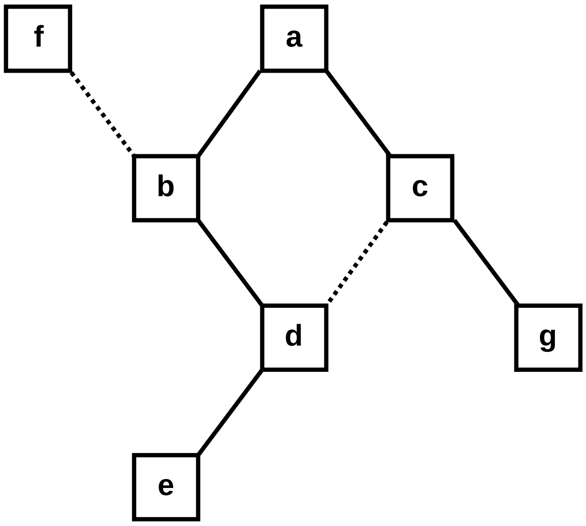

Figure 2: A Sample Autonomous System Architecture

7.3 A modification: the primary parent

Equations 10, 11, and 12 have a slight flaw: if a node has two parents with a common ancestor, the capabilities of the system will be overestimated. The d node in figure 2 is an example of this.

We solve this problem by introducing the notion of a *primary parent* for each system, and modifying the U, D, and S definitions. This has the effect of superimposing a tree structure on the DAG. In our example, the solid lines represent the primary parent/primary child link. The primary parent of a node is the only one who will incorporate updates from the child; other parents will receive the update, but they will not use it in the computation of their own update vectors. In conjunction with these definitions, we define a *primary path* as the path between two nodes using only primary links. Note that between any two nodes, only one primary path can exist.

The notions of primary ancestor, primary descendant, and primary sibling are analogous to the notions of ancestor, descendant, and sibling defined earlier. Given those definitions, we define the sets of primary parents, primary children, primary siblings, primary ancestors, and primary descendants as follows:

$$PP_x = \{p \mid primary\_parent(p, x)\} \tag{14}$$

$$PC_x = \{c \mid primary\_child(c, x)\} \tag{15}$$

$$PS_x = \{s \mid (x \neq s) \text{ and } (PP_x \cap PP_s \neq \emptyset)\} \tag{16}$$

$$PA_x = \{a \mid primary\_ancestor(a, x)\} \tag{17}$$

$$PD_x = \{d \mid primary\_descendant(d, x)\} \tag{18}$$

Equations 19, 20, and 21 incorporate the primary parent into the definitions of the U, D, and S vectors.

$$U_x = \{x\} \uplus \left(\biguplus_{\forall i \in PC_x} U_i \right) \tag{19}$$

$$D_x = \{x\} \uplus \left(\biguplus_{\forall j \in PP_x} D_j \right) \uplus \left(\biguplus_{\forall k \in PS_x} S_k \right) \tag{20}$$

$$S_x = U_x \tag{21}$$

All parents will use the U update to compute their V sets for the child (see equation 13). If a node has only one parent, it is automatically the primary parent. Conversely, a child will only incorporate the D update from its primary parent into its D vector; it will still use the D sets from its other parents to compute the V sets. Siblings that share a primary parent will incorporate updates.

$$V_{xy} = \begin{cases} U_y & y \in PC_x \\ D_y & y \in PP_x \\ \{x\} & x = y \\ S_y & y \in PS_x \\ \emptyset & \text{otherwise} \end{cases} \tag{22}$$

7.4 Proofs of semantics

In this section, we will prove that the semantics we have defined are correct. Correct semantics ensure that an autonomous system is represented at most once in any update vector. Semantics meeting this constraint will not overestimate system resources.

Lemma 1 $(y \in U_x) \Leftrightarrow (y = x \;\text{ or }\; y \in PD_x)$

> Proof (by induction): Recall the definition of U_x in equation 19. First we will prove the $\Rightarrow$ implication.
>
> > At the leaf, $U_x = \{x\}$. This fulfills the first clause of the implication, as the single member of the set is x, and therefore the implication holds. This is the base case for the proof by induction.
> >
> > For a non-leaf system, the induction step assumes that the implication is true for all primary children of x. This means that,

for each child c of x, U_c contains all descendants of c, and c itself. The first term adds x, which matches the first clause of the implication. second term of equation 19 adds all the primary children of x and their descendants. All of these must, by the definition of primary child and primary descendant, be 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 $y \in PD_x \Rightarrow y \in U_x$.

At a leaf node there are no descendants, so the base case is proven.

Once again, for an internal node, we assume that the implication is true for all its primary children, i.e. $\forall c \in PC_x$, $y \in PD_c \Rightarrow y \in U_c$, and that $c \in U_c$. Therefore, the second term adds all primary children of x, and all of their primary descendants, which is to say it adds all the primary descendants of x. Thus, $y \in PD_x \Rightarrow y \in U_x$. □

Corollary 1 $(y \in S_x) \Leftrightarrow (y = x \;\text{ or }\; y \in PD_x)$

Lemma 2 $\forall k \in PS_x, U_x \cap S_k = \emptyset$

Proof (by contradiction): Assume $\exists y \in (U_x \cap S_k)$ for some $k \in PS_x$

From lemma 1, $y \in U_x \Rightarrow y = x$ or $y \in PD_x$.

By the definition of S_k, $y \in S_k \Rightarrow y \in U_k$.

Again from lemma 1, $y \in U_k \Rightarrow y = k$ or $y \in PD_k$.

By the definition of primary siblings, x and k are both primary children of some parent, p. This implies that y has two primary paths to p, one through x, and one through k. By definition, this cannot be, and so the assumption is false.

Therefore, the lemma is true. □

Corollary 2 $\forall k \in PS_x, x \notin S_k$

Corollary 3 $\forall k \in PS_x, U_x \cap U_k = S_x \cap U_k = \emptyset$

Corollary 4 $\forall k, j \in PC_x, k \neq j \Rightarrow U_j \cap U_k = S_j \cap S_k = S_j \cap U_k = \emptyset$

Lemma 3 $y \in D_x \Rightarrow y \notin PD_x$

Proof (by induction):

At the root, $D_x = \{x\}$. By definition, $x \notin PD_x$.

For the induction step, assume that the lemma is true for all parents of the node x. Suppose that $\exists y \in (D_x \cap PD_x)$.

y cannot come from the first term in the definition of D_x, as $x \notin PD_x$. y cannot come from the second term, as that would violate the induction assumption. From lemma 2, we know that y cannot come from the third term. Therefore, y cannot exist, and the lemma is true for x in the inductive step. □

Theorem 1 (No system is represented more than once in a U vector)
$\forall y \in U_x, y \notin (U_x - y)$

Assume $\exists y \ni y \in (U_x - y)$. Then, based on equation 19, either $y = x$ and $x \in U_c$ for some $c \in PC_x$, or $U_{c1} \cap U_{c2} \neq \emptyset$ for some $c1, c2 \in PC_x, c1 \neq c2$.

From lemma 1, if $x \in U_c$, then x must be a primary descendant of itself, which is not allowed. From corollary 4, $U_{c1} \cap U_{c2} = \emptyset$.

Therefore, y cannot exist, and the theorem is proven. □

Corollary 5 (No system is represented more than once in a S vector)
$\forall y \in S_x, y \notin (S_x - y)$

Theorem 2 (No system is represented more than once in a D vector)
$\forall y \in D_x, y \notin (D_x - y)$

Assume $\exists y \ni y \in (U_x - y)$. Then, based on equation 20, one of the following must be true:

1. $x \in D_p$ for $p \in PP_x$
2. $x \in S_k$ for some $k \in PS_x$
3. $S_j \cap S_k$ for some $j, k \in PS_x, j \neq k$
4. $D_p \cap S_k$ for $p \in PP_x$ and some $k \in PS_x$

By lemma 3, (1) cannot be true. Corollaries 2 and 4 eliminate cases (2) and (3). For (4) to hold, a system simultaneously be a primary descendant of p (corollary 1 and the fact that $k \in PC_p$) and also not be a primary descendant (lemma 3). Therefore, (4) cannot hold.

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

Therefore, by theorems 1 and 2, and corollary 5, we have proven that the semantics for combining update vectors will not overestimate system resources.

8 Concluding Remarks

We have described a distributed, hierarchical scheduling system for autonomous systems. Supporting scheduling in autonomous, heterogeneous systems is a difficult task. Because information about an autonomous system might not be exported, external schedulers might have to make decisions based on incomplete information. If we want our systems to be scalable, we must condense the information that describes a system so that the size of an update message does not grow in relation to the number of processors in the system. The heterogeneity of the system introduces difficulties in the transfer of tasks, and makes the description of a system more complex.

We believe the work presented here will support many applications. Scientists will be able to use a heterogeneous group of machines to solve complex computational problems, idle workstations can be harnessed to run jobs, and research groups will be able to combine their resources to solve problems in ways not possible before.

As part of our continuing work, we are using a prototype implementation with simulation studies to determine the viability of our approach. We will also augment the command interfaces for both users and administrators to increase their expressive power, and are investigating the internalization and automation of some administrative functions.

References

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

[2] Bradley Norman Babin. DCS: A System for Distributed Computation. Master's thesis, Oregon State University, May 1988.

[3] K. Birman, R. Cooper, T. Joseph, K. Marzullo, M. Makpangou, K. Kane, F. Schmuck, and M. Wood. *The ISIS System Manual, Version (2.1)*, September 1990.

[4] Ben A. Blake. Assignment of independent tasks to minimize completion time. *Software–Practice and Experience*, 22(9):723–734, September 1992.

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

[6] Mic Bowman, Larry L. Peterson, and Andrey Yeatts. Univers: An attribute-based name server. *Software–Practice and Experience*, 20(4):403–424, April 1990.

[7] Michael J. Carey, Miron Livny, and Hongjun Lu. Dynamic Task Allocation in a Distributed Database System. In *Distributed Computing Systems*, pages 282–291. IEEE, 1985.

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

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

[10] S. Deering. Host Extensions for IP Multicasting. RFC 1054, Stanford University, May 1988.

[11] A.L. DeSchon and R.T. Braden. Background file transfer progrm (bftp). RFC 1068, Network Information Center, August 1988.

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

[13] Brent F. Dubach, Robert M. Rutherford, and Charles M. Shub. Process-Originated Migration in a Heterogeneous Environment. In *Proceedings of the Computer Science Conference*. ACM, 1989.

[14] 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.

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

[16] 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.

[17] 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.

[18] Christopher A. Gantz, Robert D. Silverman, and Sidney J. Stuart. A Distributed Batching System for Parallel Processing. *Software–Practice and Experience*, 1989.

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

[20] Simson Garfinkel and Eugene Spafford. *Practical Unix Security*. O'Reilly and Associates, 1991. ISBN 0-937175-72-2.

[21] Leo Geurts, Lambert Meertens, and Steven Pemberton. *ABC Programmer's Handbook*. Prentice Hall, 1990. ISBN 0-13-000027-2.

[22] Dorit Hochbaum and David 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.

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

[24] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

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

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

[27] Virginia Mary Lo. Task Assignment to Minimize Completion Time. In *Distributed Computing Systems*, pages 329–336. IEEE, 1985.

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

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

[30] Sape J. Mullender. Process Management in a Distributed Operating System. In J. Nehmer, editor, *Experiences with Distributed Systems*, pages 38–51. Springer-Verlag, 1987.

[31] 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.

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

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

[34] J.B. Postel. File transfer protocol. RFC 768, Network Information Center, August 1980.

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

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

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

[38] Sequent. Sequent hardware manuals.

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

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

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

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

[43] 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.

[44] 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.

[45] David Velten, Robert Hinden, and Jack Sax. Reliable Data Protocol. RFC 908, Network Information Center, July 1984.

[46] Larry Wall and Randal L. Schwarz. *Programming* **perl**. Nutshell Handbook. O'Reilly & Associates, 1990. ISBN 0-937175-64-1.