Purdue University

Purdue e-Pubs

Department of Computer Science Technical Reports | Department of Computer Science

1991

Scheduling Support for an Internetwork of Heterogeneous, Autonomous Processors

Steve Chapin

Eugene H. Spafford
Purdue University, spaf@cs.purdue.edu

Report Number:
92-006

Chapin, Steve and Spafford, Eugene H., "Scheduling Support for an Internetwork of Heterogeneous, Autonomous Processors" (1991). *Department of Computer Science Technical Reports.* Paper 931.
https://docs.lib.purdue.edu/cstech/931

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries.
Please contact epubs@purdue.edu for additional information.

SCHEDULING SUPPORT FOR AN INTERNETWORK OF HETEROGENEOUS, AUTONOMOUS PROCESSORS

Steve Chapin
Eugene H. Spafford

CSD-TR-92-006
January 1992

Scheduling Support for an Internetwork of Heterogeneous, Autonomous Processors

Technical Report TR-CSD-92-006

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

January, 1992

Abstract

We are investigating support for distributed, hierarchical scheduling of tasks on autonomous, heterogeneous computing systems. Many researchers have studied the related problem of determining near-optimal task placement in systems possessing some of these attributes. Their algorithms assume the existence of mechanisms to gather information about the system, move tasks, and perform other related operations. We are attempting to define and construct these mechanisms in an abstract fashion that allows them to be generalized to other distributed architectures.

Autonomous systems consist of one or more subsystems connected by a message-passing communications medium; at the lowest level, a processor is an autonomous system with no subsystems. Processors within the same system may be of different types. All information, behavior, and policy pertaining to an autonomous system is private and local to that system. Any sharing of this information is at the discretion of the local system.

The goals of our research are to provide scalable mechanisms for efficient implementations of scheduling policies on systems ranging from a few workstations on a local-area network to thousands of machines spread over a large geographical area and connected by arbitrary interconnection links. We are examining the requirements of these systems for different levels of security, reliability, load sharing, and location transparency. We hope to be able to characterize these and other properties in a scalable mechanism that does not impose undue load demands while supporting a wide range of distributed scheduling policies.

1 Introduction

This paper describes work being done under the MESSIAHS[1] project. We propose to define and build distributed, hierarchical scheduling mechanisms for autonomous, heterogeneous systems. We will first define some necessary terms, and then we will give background information and describe the scheduling support mechanisms we intend to build.

Autonomous systems consist of one or more subsystems connected by a communications 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.

Our systems are *distributed*, which communicate by passing messages over an external communications channel. Such systems are often called multicomputers (as defined in [12]) or loosely-coupled systems, as opposed to tightly-coupled parallel machines that communicate through shared memory.

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, although they may vary in performance.

There are many examples of systems that share some of these qualities. Shared-memory parallel processors such as the Sequent Symmetry are homogeneous, tightly-coupled systems. Networks of workstations such as those described in [1, 7, 8] are homogeneous and loosely-coupled. [13] describes a heterogeneous, distributed system of workstations. None of these systems support autonomy.

It is vital that a system for distributed computation support autonomy because of the prevailing decentralization of computing resources. There is usually no longer a single, authoritative controlling entity for the computers in a large organization. A scientist may control a few of his own machines, 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 national organization. No single entity, from the scientist to the national organization, has complete control over all the computers it may wish to use.

A feature of autonomy that affects long-lived jobs is revocation. For example, a computation might start on an unused workstation at night. If the local scheduling policy determines that the job would no longer be run, the acceptance of the program must be revoked, and it would have to be migrated within the system. This might occur if the user returned to his workstation in the morning, or if the load average rose above a certain threshold. Efficient implementations of revocation would use mechanisms such as checkpointing and process migration. These mechanisms can also be used later in developing fault tolerance and load balancing schemes. Without

[1] MEchanisms for Scheduling Support In Autonomous Heterogeneous Systems.

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.

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

Within a distributed system, there are two levels of scheduling: the association of tasks with processors, also called task placement, and the choice of which task to make ready on a given processor. Our work concentrates on facilities for the former. Our definition of a *task* includes the conventional model of a computationally intensive unit in a larger program, as well as a set of database queries (see [4]), 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 efficiently, or to allocate network resources for large data transfers. For simplicity of description, we will use the conventional model of scheduling tasks on processors.

Many researchers have studied the related problem of task placement in distributed systems; however, their algorithms have assumed either that all processors are similar, or that they have total control of all processors in the system. We have found no evidence in the literature that the 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.

An important distinction must be drawn between the scheduling support mechanisms and the scheduling policies (and associated algorithms) the mechanisms support. 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 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.

The remainder of this paper describes the goals for our scheduling mechanism (§2), the system architecture model (§3), our proposed approach to solving the problem of supporting distributed scheduling (§4), an example to illustrate our model (§5), our areas of planned research (§6), and our conclusions (§7).

2 Goals and Assumptions

We have five goals for our scheduling mechanism:

efficient scheduling The mechanisms must support scheduling algorithms in an efficient manner. 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 will measure the efficiency of our support mechanisms through simulations to determine the relative difference between the schedule produced by an algorithm using our mechanisms, and the same algorithm running with perfect information.

scheduling 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.

scalability The architecture should work on systems ranging from a single workstation to thousands of processors, with interconnection schemes ranging from local area networks to wide area networks.

noninterference The monitoring overhead and message traffic must be kept low so as to not adversely impact performance within the system.

extensibility Since 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.

We have made several assumptions when designing the system. Our work builds on earlier work that defined mechanisms for task migration (see [9]) and architecture independent task and data representations (see [6]). Any machine that submits tasks to the system must reciprocate by accepting scheduling requests, but not necessarily honor them. The makeup of the system is dynamic; machines are free to leave or join the system at any time.

Equally important is what we do not assume. We do not assume that communication between systems is reliable, or that a subsystem belongs to only one autonomous system.

3 The Architecture Model

We intend to build our autonomous systems in a hierarchical fashion, which is a model often used in the real world. For example, the Internet is composed of many autonomous systems, one of which is the set of computers at Purdue University. Within the Purdue hierarchy, there are many subordinate autonomous systems, including those used by the School of Engineering (ecn), the Department of Computer

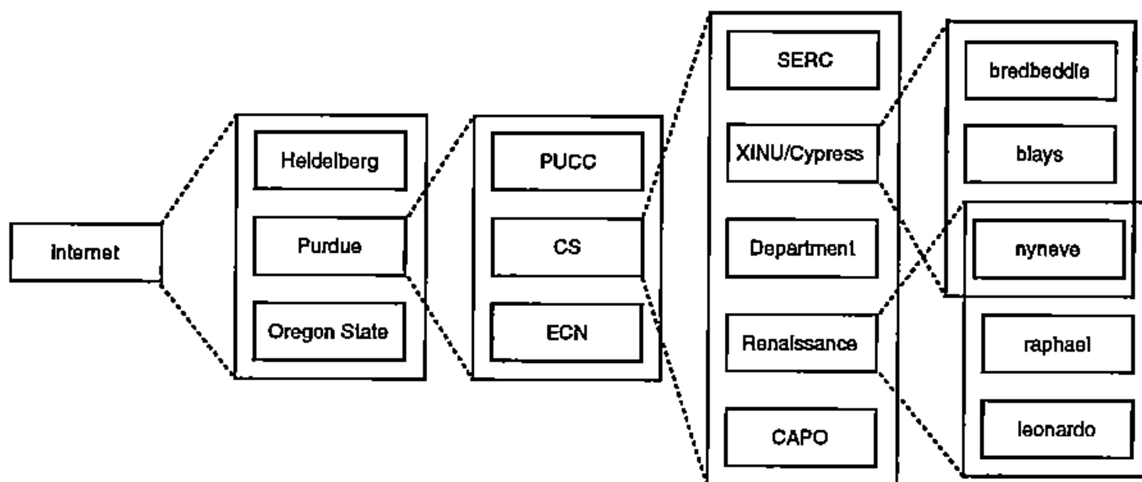

Figure 1: A Sample Autonomous System Architecture

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. Within each of these sets there are many machines, which could be further grouped into autonomous systems. At the lowest level, each machine can be viewed as an autonomous system. This is pictorially represented in figure 1.

Note that 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. For example, 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; therefore it belongs to two autonomous systems.

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 system may have multiple parents.

4 The MESSIAHS Approach

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.[2] 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*, the *task description vector*, and the protocol used to communicate between systems.

4.1 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 subordinate systems.

At this time, the system description vector is designed to support the scheduling of conventional tasks. A properly designed mechanism will allow us to tailor the vector to the application, e.g. the distributed database mentioned earlier.

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. We augmented the set with items we expect will be desired in the future. Factors in this set include:

- memory statistics (available and total)
- processor load (queue length and percentage of busy time)
- special services (I/O devices, vector processors, etc.)
- 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 will have the ability to cache information to build a history of behavior for subordinate systems. This history mechanism will be user-configurable, and information on the history characteristics of each datum (e.g. permanence, mean, standard deviation, etc.) will be passed with the datum. The representation of these data is an open problem and is one of our current areas of investigation.

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

4.2 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.

- memory requirements
- estimated run time
- originating system
- special services required
- estimated communications load

4.3 The protocol

The communications protocol defines the interaction between scheduling modules within the autonomous system. All information passing and inter-module coordination takes place through the protocol.

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.

4.3.1 The update channel

The update protocol is message based. Each message contains the system description vector for the autonomous 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 in [3], networks are generally reliable under normal use.

At periodic intervals a module will recompute its status vector and advertise the updated vector through the update channel. The length of the period is a locally tunable parameter. 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 in [2, 5]. 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. 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. In this way, we have placed no restrictions on the communications structure within an autonomous system; it may range from being nonexistent to being a fully connected graph within the hierarchy.

4.3.2 The control channel

The control channel is intended to be a bidirectional, reliable, message based channel. Although reliable message passing has been a topic of research (see [10, 14]), there does not exist an accepted and widely implemented standard for reliable message passing. Therefore, we will encapsulate our messages in a reliable stream protocol such as TCP [11]. 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 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 identicfication number of the rejected SCHED_REQUEST message.

TASK_REQUEST The system requests a task from another system (used in receiver-initiated load balancing schemes). This request includes a copy of a task description vector describing tasks the requester will accept.

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.

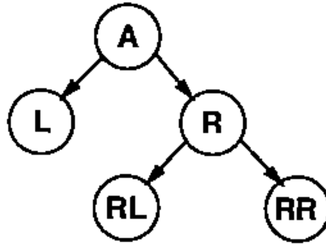

Figure 2: A simple autonomous system hierarchy

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 ID number of the rejected TASK_REQUEST message.

TASK_REVOKE A system that has been executing a task is no longer willing to do so.

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

4.3.3 The task channel

We intend to use a reliable streaming protocol to implement the task channel. Although a task is conceptually a single message, and thus could be sent within a reliable message protocol, this is not feasible for two reasons. First, as noted previously, there is not a standard facility of this type available. Second, message size limits in existing protocols are typically about 64 kilobytes, which would force a large task to be fragmented into several messages and reassembled.

The task channel 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.

5 Example

We now present a simple example of how our mechanisms could be used. In this example, X_m denotes the scheduling module that implements our mechanism, running on system X. X_p denotes the program that implements the scheduling policy.

Consider the simple hierarchy displayed in figure 2. Let us assume that the update period for each system is one minute, so that every minute, each system evaluates its

system state and sends an update message to its parent. Thus, RR_m and RL_m send updates to R_m. R_m and L_m send updates to A_m. Depending on the timing of the update messages, R_m may use outdated information about RR or RL when computing the description vector to send to A_m.

Suppose that a user submits a job composed of a single task to R_p. R_p compares the requirements of the task against the descriptions of capabilities it has for itself and its children. It determines that it will not take the task, but finds that, according to the most recently received updates, RL is capable of handling the task, and RR is not. R_p sends a SCHED_REQUEST control message to RL_p. RL_p decides to accept the task, so RL_p sends a SCHED_ACCEPT control message to R_p. R_p then opens a task channel to RL_p and uses it to transfer the task. If RL_p had not accepted the task, it would send a SCHED_DENY message to R_p. R_p would send a SCHED_REQUEST message to A_p, because no node within R's autonomous system would accept the task.

When the computation of the task is partially completed, RL_p determines that the local policy dictates that RL will no longer work on the task. It suspends the task and sends a TASK_REVOKE control message to R_p. R_p concludes that it has no children eligible for the job, and passes a SCHED_REQUEST message to A_p.

A_p determines it cannot take the task. L_p has the capability to process the task, so A_p passes a SCHED_REQUEST message through the control channel to L_p. L_p accepts the task, and a task channel is opened between RL_p and L_p to transfer the task.

6 Research Areas

We are investigating three areas of research that will lead to a prototype implementation of MESSIAHS. Each area has its own questions that must be answered. The areas are:

description vectors The fixed content of the system and task description vectors must be set. To determine the fixed content, we surveyed existing scheduling algorithms and are analyzing them to find similarities and common requirements.

protocol design This requires determining the information each channel carries, and its format. Particular issues that must be addressed are the representation of the task and system vectors, and the formalization of the control messages. We will examine the possibility of extending the update channel to allow parents to pass update messages to children without causing update cycles.

programmer's interface We will have to specify the interface to the scheduling support module for implementers of scheduling algorithms.

7 Conclusions

We have described a distributed, hierarchical scheduling system for autonomous systems. We will be completing the design and using a prototype implementation with simulation studies to determine the viability of our approach.

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

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; research groups will be able to combine their resources to solve problems in ways not possible before.

References

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

[2] 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.

[3] 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.

[4] 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.

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

[6] 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.

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

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

[9] 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.

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

[11] J.B. Postel. DoD standard Transmission Control Protocol, January 1980. RFC 761.

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

[13] 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.

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