

# A Brief Examination of Distributed Scheduling Support for Autonomous Systems

Steve J. Chapin\*  
Dept. of Math. & Computer Science  
Kent State University  
Kent, OH 44242-0001  
sjc@cs.kent.edu

## Abstract

MESSIAHS is a set of mechanisms supporting task placement in heterogeneous, distributed, autonomous systems. MESSIAHS provides a substrate on which scheduling algorithms can be implemented. The chief design goals for MESSIAHS include support for autonomy, flexibility, scalability and efficiency. These mechanisms were designed to support diverse task placement and load balancing algorithms.

This paper focuses on MESSIAHS's support for the autonomy of constituent nodes within distributed systems. We define four types of autonomy, show how they influence distributed scheduling, and explain how the mechanisms support each type of autonomy.

Keywords: scheduling, autonomy, distributed computing

## 1 Introduction

Continuing advances in microprocessor technology, and the commensurate increases in computational power of microprocessor-based workstations, have caused increasing attention to be focused on building large-scale distributed computing systems. At the same time, existing workstation computing facilities are often underutilized [8, 10]. The chief reason that users do not allow their workstations to be used for distributed computation is their fear that they will lose control of their local resources [8].

A vital tool for effective utilization of distributed systems is task placement, or scheduling.<sup>1</sup> Task scheduling chooses one processor from a set of available processors to run a task [1].

---

\*This work was sponsored in part by NASA GSRP grant number NGT 50919.

<sup>1</sup>A *task* or *process* is a program to be run by the distributed system.

The MESSIAHS<sup>2</sup> system [2, 3] provides a set of mechanisms that facilitate global scheduling in distributed, heterogeneous, autonomous systems. In this paper, we will focus on aspects of MESSIAHS that preserve the autonomy of nodes within the system. Autonomy manifests itself in large distributed systems because no single entity has administrative authority over all machines within the system.

We define distributed systems as those that communicate via message passing. Heterogeneous systems may have different instruction set architectures, data formats, and attached devices. Autonomous systems make all policy decisions locally. Subsequent uses of the terms *distributed system* or *system* refer to a distributed, autonomous, heterogeneous system. We use the terms *processor*, *node*, and *machine* synonymously, to refer to an individual entity within an autonomous system.

The next section defines four types of autonomy, and indicates their impact on distributed systems. Section 3 describes the class of distributed systems for which our work is applicable, and outlines features of MESSIAHS that support the four types of autonomy. The final section gives concluding remarks.

## 2 Autonomy in Distributed Systems

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

---

<sup>2</sup>Mechanisms Effecting Scheduling Support In Autonomous, Heterogeneous Systems.

is at the discretion of the local system.

Garcia-Molina and Kogan [9], and Eliassen and Veijalainen [6] have examined autonomy in distributed systems and devised taxonomies for different types of autonomy. The scheme proposed by Eliassen and Veijalainen is more general but less detailed than that proposed by Garcia-Molina. The following four classes of autonomy combine the two schemes, tailor the definitions to the application of distributed scheduling, and define a new type of autonomy.

#### design autonomy

The designers of individual systems are not bound by other architectures, but can design their hardware and software to their own specifications and needs. Design freedom can lead to heterogeneity.

#### communication autonomy

Separate systems can make independent decisions about what information to release, what messages to send, and when to send them. A system is not required to advertise all its available facilities, nor is it required to respond to messages received from other systems.

#### administrative autonomy

Each system sets its own resource allocation policies, independent of the policies of other systems. In particular, a local system can run in a manner counterproductive to a global optimum.

#### execution autonomy

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

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

To be considered autonomous, a system must display some degree of all four types of autonomy. In turn, mechanisms supporting task placement in autonomous systems must support all four types of autonomy. Therefore, the mechanisms must run on multiple architectures, allow local decisions regarding communication with external systems and execution of tasks, and support a local scheduling policy.

Because of execution and communication autonomy, all decisions pertaining to a system are under its control. The system advertises as little or as much of its system state as its local policy decrees, and cannot be forced to accept tasks for execution. Therefore,

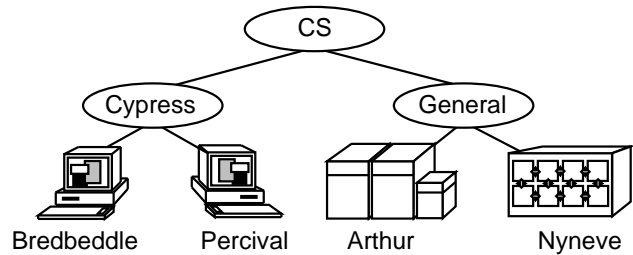


Figure 1: a sample distributed system

a machine  $A$  may not receive complete information describing machine  $B$ ;  $A$  knows only what  $B$  chooses to tell  $A$ .

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

The combination of communication and design autonomies with execution autonomy poses another problem for process migration. Execution autonomy can require the migration mechanisms to move a process from one machine to another, but because of communication autonomy and design autonomy, the sender may not know the architecture of the recipient machine. Therefore, advance translation of the program image might be impossible. Machines with different instruction sets cannot directly share code. Schemes such as those proposed in Essick [7] and Shub [5] for machine-independent program representation might alleviate this problem.

### 3 Autonomy Support in MESSIAHS

MESSIAHS supports task placement in distributed systems with hierarchical structure based on administrative domains, modeled by directed acyclic graphs. This structure arises from existing computing systems. Multiple subordinate systems can be combined into an encapsulating system, yielding the hierarchical structure. The nodes of the graph represent the autonomous systems, and edges indicate encapsulation.

Figure 1 shows an example distributed system based on the Kent State University Department of Mathematics and Computer Science. In the example, the Mathematics and Computer Science department contains two administrative domains, HOSS and General. HOSS in turn encapsulates the research machines

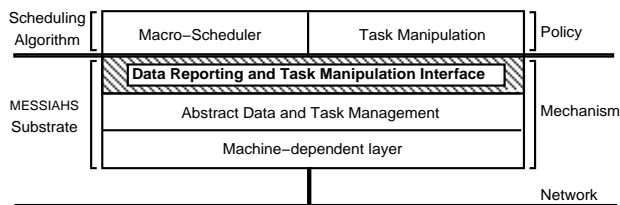


Figure 2: structure of a MESSIAHS scheduling module

belonging to the HOSS project, and General contains the general purpose servers for the department.

### 3.1 Scheduling Modules

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

The system description vector encapsulates the state of a system. A scheduling module uses an SDV to advertise its abilities to other systems that may request task scheduling. Scheduling modules use SDVs as the basis for choosing a candidate system for a task from among their neighboring systems.

The description vector contains a fixed portion and a variable portion. The fixed portion contains data items supporting the scheduling algorithms from the literature. The variable portion supports administrative autonomy by permitting the customization of the information set in support of specialized policies.

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

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

Figure 2 displays the structure of a MESSIAHS

scheduling module. The machine-dependent layer handles raw data exchange between scheduling modules, collects the local state information, and interacts with the task manipulation mechanisms specific to the local operating system. The abstract data and task management layer provides an abstract interface for the machine-dependent operations to the data reporting layer. The third layer, data reporting and task manipulation, presents the administrator with the interface to the MESSIAHS mechanisms. The administrator supplies the topmost layer, which embodies the scheduling policy for the system. All three of the lower layers provide some support for autonomy.

It is obvious that all four types of autonomy cannot be supported to their fullest. If they were, any scheduling that occurred would be by blind luck. Therefore, the MESSIAHS mechanisms concentrate on supporting autonomy to the greatest practical degree, while still facilitating scheduling.

### 3.2 The Machine-dependent Layer

The machine-dependent layer fulfills four functions: information encoding, access to network and transport protocols, data acquisition, and task manipulation. The lowest layer abstracts these machine-dependent features and presents them to higher layers, thereby supporting design autonomy of both the underlying hardware and operating system software. Information encoding uses an external data representation, and the data acquisition routines use operating system-specific calls to obtain system state information to fill the SDV.

The set of task manipulation primitives contains six members that support execution autonomy: *start*, *kill*, *suspend*, *resume*, *checkpoint*, and *migrate*. *Start* begins execution of a program image as a task. *Kill* aborts a running task. *Suspend* temporarily stops a running task, and *resume* restarts a suspended task. *Checkpoint* saves a task to a program image, and *migrate* moves a program image between machines. The current MESSIAHS prototype provides support for, but does not include, migration of architecture-dependent processes between heterogeneous systems.

### 3.3 Abstract Data and Protocol Management

The middle layer provides a uniform implementation of primitives for inter-module communication. It consists of a set of event-based semantics that define inter-module interaction, the communication proto-

cols used by the modules, and the extension mechanism for the description vectors.

### 3.3.1 Event-based Semantics

The support mechanisms use event-based semantics. Figure 3 depicts the hierarchy of events. There are three types of events: *finished events*, *timeout events*, and *message events*. Each event has an associated *handler*, which performs actions in response to an occurrence of the event. The handlers can be customized to implement the scheduling policy, thus supporting administrative autonomy.

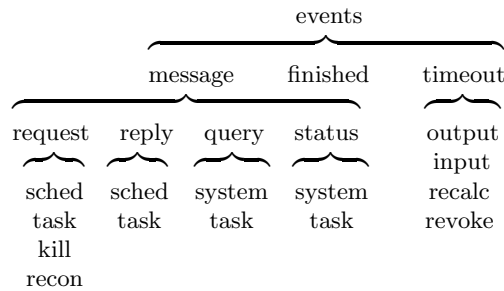


Figure 3: hierarchy of events

We now describe the events, along with typical behavior of the corresponding handlers. A finished event occurs when a task completes execution on the local host. The finished event handler notifies the originating system that the task has completed, and returns any results.

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

Message events occur when a message arrives for a module. There are four classes of message events: *request message events*, *reply message events*, *query message events*, and *status message events*. Each of these message event types has subtypes. Request message events ask the handler to perform a service, and comprise *schedule request message events*, *task request*

*message events*, *kill request message events*, and *re-configuration request message events*. Reply messages occur in response to request messages, and reply message events have two subtypes: *schedule reply message events* and *task reply message events*. Reply message events are paired with the corresponding request message event subtypes.

Query message events and status message events have two subtypes, *task* and *system*. Query events ask the handler to provide for information about tasks and systems, rather than for services to be performed. Status messages contain information describing tasks and systems, and status message events may occur without any query taking place.

## 3.4 The Protocols

The communication protocols define the interaction between scheduling modules within the distributed system. All information passing and inter-module coordination takes place through the protocols. Conceptually, the protocol has three channels: the control, update, and task channels. The update channel advertises system state. The task channel moves a task between systems, and the control channel is used to pass control messages and out-of-band data.

The communications protocols have several relevant features. First, they are designed to support communication autonomy, in that the recipient of a message is not required to respond to it. Second, the task channel supports a mode of transfer called *proxy transfer*. Proxy transfer allows an encapsulating system to act as an intermediary between an internal node and an external node, thus hiding the internal node from external observation. Third, the protocols directly support the message events from figure 3.

## 3.5 The Extension Mechanism

It is impossible to predefine the complete set of characteristics used by all present and future scheduling algorithms. Therefore, the description vectors include an extension mechanism that allows users to customize the description of a system or task. Users may append a set of simple values to the description vector, in the form of `(type, variable, value)` triples.

## 3.6 The Interface Layer

The interface layer is the implementation vehicle for the scheduling policy, and allows administrators to express policy in terms of the operations provided by the management and machine-dependent layers. It is

here that the event handlers are implemented. Under this system, program distribution is under the control of the autonomous system, and therefore the administrators, rather than the programmer.

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

Two interface layers have been implemented, and are described in [2]. The first layer is a library of function calls. The second layer is an interpreter for a simple language specifically designed to support task scheduling.

## 4 Concluding Remarks

This paper described the importance of autonomy support in distributed systems, and outlined four types of autonomy: *execution autonomy*, *communication autonomy*, *design autonomy*, and *administrative autonomy*.

We examined the impact of these four types of autonomy on scheduling support for distributed systems. In addition, we examined the structure of the MESSIAHS scheduling module and its features that support task scheduling in the presence of autonomy in distributed systems. This support includes preservation of each of the four types of autonomy for the machines comprising the distributed system.

These features are summarized in the following list:

- a layered scheduling module that supports design autonomy for both hardware and software,
- events and their associated handlers to provide administrative autonomy,
- two interface layers that provide access to underlying autonomy support,
- extensible description vectors that provide flexibility and support administrative autonomy,
- a request/reply paradigm supporting execution autonomy, and
- proxy acceptance to support communication autonomy.

In summary, the MESSIAHS mechanisms support distributed scheduling in the presence of autonomy. As proof of concept, a set of scheduling algorithms

that spans the taxonomy in [1] was implemented using the MESSIAHS prototype, and its performance measured [3]. Readers interested in further information on MESSIAHS should contact the author, or see also [2, 4].

## References

- [1] T. L. Casavant and J. G. Kuhl. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. *IEEE Transactions on Software Engineering*, 14(2):141–154, February 1988.
- [2] S. Chapin and E. Spafford. Implementing Scheduling Algorithms Using MESSIAHS. *Scientific Programming*, 1994. to appear in a special issue on Operating System Support for Massively Parallel Computer Architectures.
- [3] S. J. Chapin. Scheduling Support Mechanisms for Autonomous, Heterogeneous, Distributed Systems. Ph.D. Dissertation, Purdue University, 1993.
- [4] S. J. Chapin and E. H. Spafford. Constructing Distributed Schedulers with the MESSIAHS Interface Language. In *27th Hawaii International Conference on Systems Sciences*, volume 2, pages 425–434, Maui, Hawaii, January 1994.
- [5] B. F. Dubach, R. M. Rutherford, and C. M. Shub. Process-Originated Migration in a Heterogeneous Environment. In *Proceedings of the Computer Science Conference*. ACM, 1989.
- [6] 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.
- [7] R. B. Essick IV. *The Cross-Architecture Procedure Call*. PhD thesis, University of Illinois at Urbana-Champaign, 1987. Report No. UIUCDCS-R-87-1340.
- [8] C. A. Gantz, R. D. Silverman, and S. J. Stuart. A Distributed Batching System for Parallel Processing. *Software-Practice and Experience*, 19, 1989.
- [9] H. Garcia-Molina and B. Kogan. Node Autonomy in Distributed Systems. In *ACM International Symposium on Databases in Parallel and Distributed Systems*, pages 158–166, Austin, TX, December 1988.
- [10] M. J. Litzkow. Remote UNIX: Turning Idle Workstations Into Cycle Servers. In *USENIX Summer Conference*, pages 381–384, 2560 Ninth Street, Suite 215, Berkeley, CA 94710, 1987. USENIX Association.