

Distributed Scheduling Support in the Presence of Autonomy

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.

1 Introduction

Continuing advances in microprocessor technology, and the commensurate increases in computational power of microprocessor-based workstations and multiprocessors, 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 [6]. 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 [6].

To maximize the utilization of new and existing machines, while simultaneously respecting the wishes of the owners of those machines, it is necessary to build distributed systems that respect the autonomy of their constituent members. Our solution to this problem is to collect separate processors into a distributed system, and to recursively join the distributed systems into larger systems to

further expand the computational power of the whole.

A vital tool for effective utilization of distributed systems is task scheduling.¹ Task scheduling has two components: global scheduling and local scheduling [2]. Global scheduling, also known as task placement, chooses one processor from a set of available processors to run a task. A processor uses local scheduling to choose which of its assigned tasks to run next. All further uses of the term *scheduling* in this paper refer to task placement.

The MESSIAHS² system 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. 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.

The systems under consideration display all three attributes, connecting machines of differing architectures and with individual administrative authorities via a communications network. 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. Thus, our definition of system includes a single machine, a pair of homogeneous workstations communicating via a local-area network, or a system having thousands of nodes, including personal computers, workstations, parallel processors, and supercomputers, residing at several remote sites and connected by a wide-area network.

Heterogeneity can provide efficient and cost-effective methods for performing certain computations. A large computation might have certain pieces best suited for exe-

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

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

²Mechanisms Effecting Scheduling Support In Autonomous, Heterogeneous Systems

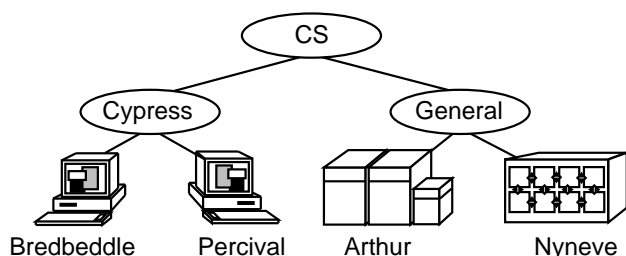


Figure 1: a sample distributed system

cution on a hypercube architecture, while other parts might run best on a shared-memory parallel processor or a graphics workstation. Restricting this computation to using only one architecture will cause it needless delay. In other cases, tasks such as text processing or high-level language interpretation may be independent of any single architecture. The ability to use any available machine, regardless of architecture, can increase the throughput for the distributed system.

The next section describes the class of distributed systems for which our work is applicable, and outlines the architecture of the MESSIAHS mechanisms. Section 3 defines four types of autonomy, and indicates their impact on distributed systems. Section 4 describes features of MESSIAHS that support the four types of autonomy, while section 5 details the application of these features to the specific problem of supporting heterogeneity in distributed systems. The final section gives concluding remarks.

2 Background

MESSIAHS supports task placement in distributed systems with hierarchical structure based on administrative domains, modeled by directed acyclic graphs. This structure is inspired by existing computing systems, such as those within a typical Computer Science department (see figure 1).

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. The graph is directed downward; the edges are directed from encapsulating nodes, or parents, to subordinate nodes, or children. Children of the same parent are siblings. The *neighbors* of a system are its children, parents, and siblings.

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 belonging to the Heterogeneous Operating System Software project, and General contains the general purpose servers for the department. Ogion and Vetch are children of HOSS, and therefore are siblings.

Each component system runs a *scheduling module* that implements the local scheduling policy and manages administrative aspects of the system. These modules exchange data sets describing the status of the systems. On demand, the modules also process scheduling requests, which contain a description of the task for which the sender requests service. It should be noted that under this system, program distribution is under the control of the autonomous system, and therefore the administrators, rather than the programmer.

Each module exchanges status information only with modules running on its neighbors. Because of the hierarchical structure of the system, some nodes might be invisible to other nodes. In the example system from figure 1, Chaos receives status information only from Sequent and General, and sees no information that can be directly related to Ogion or Vetch. The capabilities of Ogion and Vetch are subsumed and combined within General's state advertisement.

3 Autonomy in Distributed Systems

We define autonomous systems as those that make local policy decisions and can act without the permission of any external authority. In autonomous systems, all information, behavior, and policy pertaining to a system are private to that system. Any disclosure of private information is at the discretion of the local system.

Other researchers [5, 7] have examined autonomy in distributed systems and devised taxonomies for different types of autonomy. The following four classes of autonomy combine the two schemes, tailor the definitions to the application of distributed scheduling, and define a new type, administrative 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, as machines can have distinct instruction sets, byte orderings, processor speeds, operating systems, and devices.

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.

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.

administrative autonomy

Each system sets its own resource allocation policies, independent of the policies of other systems. Execution autonomy allows a system to have a local scheduling policy; administrative autonomy allows the local policy to be unique. In particular, a local system can run in a manner counterproductive to a global optimum.

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.

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 moving it to a new processor (called *task migration*).

Combinations of the four types of autonomy can impose substantial obstacles to task placement. For example, 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. Finding a solution to the heterogeneous task migration problem remains an open research topic.

4 Autonomy Support in MESSIAHS

As noted earlier, each node within the distributed system runs a scheduling support module that is responsible for maintaining the set of information required by the global scheduling policy and distributing information describing the system state to its neighbors within the graph. This module also controls task execution and movement through the system, and is responsible for collecting data describing the local system. The module provides the mechanism upon which the scheduling policy is built.

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.

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

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

A state advertisement and request mechanism lies at the heart of the scheduling module. A machine advertises its state through a *system description vector* (SDV) that describes the capabilities and state of a system. When a task is to be scheduled, modules exchange *task description vectors* (TDVs) describing the resource requirements of a task. The format of both the SDV and TDV were derived as a result of a survey of the scheduling literature which included 47 scheduling algorithms, as presented in [4].

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

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

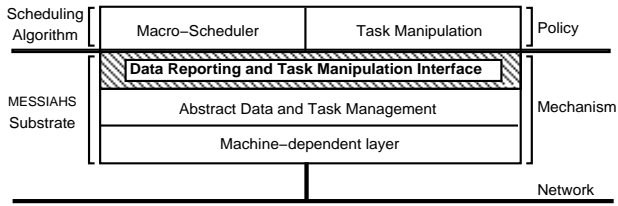


Figure 2: structure of a MESSIAHS scheduling module

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 (vectors processors, specific architectures), operating system requirements (UNIX, VMS), or software requirements (text processors, compilers).

4.1 Module Structure

The structure of a scheduling module is in figure 2. Three layers make up the module: the interface layer, the abstract management layer, and the machine-dependent layer. The machine-dependent layer implements communications protocols, task manipulation primitives and data acquisition routines over the native operating system. The abstract management layer uses the machine-dependent layer to communicate with other modules, and provides abstract, architecture and operating system independent operations for data communication and interpretation. The interface layer presents the algorithm implementer with access to the abstract operations in the management layer. MESSIAHS does not determine policy; the three layers provide mechanisms to implement scheduling policies. The interface layer is the administrator’s vehicle for expressing and implementing the local policy through the MESSIAHS mechanisms.

4.2 Machine-Dependent Layer

The machine-dependent layer fulfills four functions: information encoding, access to network and transport protocols, data acquisition, and task manipulation. The low-

est 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.

This layer provides routines to convert the machine-dependent encodings of basic data types into a standard format such as XDR [9]. 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.

4.3 Abstract Data and Task Management

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

4.3.1 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.

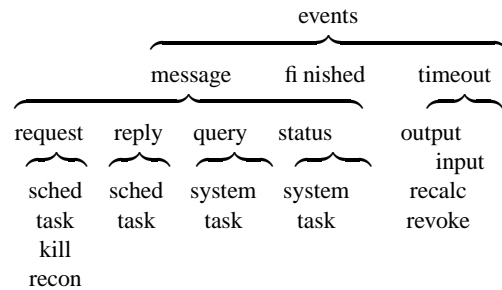


Figure 3: hierarchy of events

A finished event occurs when a task completes execution on the local host. The finished event handler notifies

the originating system that the task has completed, and returns any results.

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

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

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

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

4.4 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 be-

tween systems, and the control channel is used to pass control messages and out-of-band data. The update and control channels only connect neighbors within the distributed system; the task channel may connect any two systems. This sacrifice of communication autonomy is necessary to maintain administrative and communication autonomy at higher levels in the hierarchy.

4.4.1 Update Channel

The update protocol is message-based. Each message contains the system description vector for the sending system, and consists of a message header and a fixed set of data, followed by an optional set of policy-defined data. The interpretation of the policy-defined data is done by the two modules at opposite ends of the channel. The update channel is unidirectional; the recipient of an update message returns no information through the update channel. The update channel makes no attempt to ensure reliability.

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

With timeout-driven updates, the administrator sets the timer for output timeouts. When the countdown timer expires, the scheduling support module advertises the state representation for its system through the update channel. The length of the period is a locally tunable parameter, granting more administrative autonomy to the system.

4.4.2 Control Channel

The control channel is intended to be a bidirectional, reliable, message-based channel, such as the Simple Reliable Message Protocol [8]. A control message consists of a header, including an ID number for the message and a message type, and data that depends on the type of the message. The following defined control message types correspond to message events: *request messages*, *reply messages*, *query messages*, and *status messages*. Each of these message types has subtypes, and there is a direct correspondence between the message event subtypes depicted in figure 3 and the control message subtypes.³

It is important to note that the request/reply message pairs support execution autonomy, because no node is ever required to take an action. For example, the recipient of a kill request is not obligated to terminate a running task. In addition, communication autonomy permits the recipient to forgo informing the requester if the request was honored. While the recipient will typically respond, it is not required to do so.

³The only exception is the “system status” message event, which corresponds to an SDV arriving through the update channel.

4.4.3 Task Channel

The task channel reliably transfers a task between two nodes in a distributed system. After negotiating a task's destination through the control channel, the module opens a task channel to move the task. This may either be directly between the source and destination, or by a special form of delivery called *proxy transfer*. Proxy transfer is used to support communication autonomy, when the destination is inside an encapsulating system that prohibits outside systems from directly accessing its members. In this case, the task is delivered to the encapsulating system, which is then responsible for forwarding the task to its destination.

4.5 Extension Mechanism

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

4.6 Interface Layer

The interface layer is the implementation vehicle for the scheduling policy, and allows administrators to express policy in terms of the operations provided by the management and machine-dependent layers. The chief function of the interface layer are to make lower-level capabilities available to the administrator in the support of administrative autonomy. It is here that the event handlers are implemented.

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

Two interface layers are described in [3]. The first is a library of function calls. An administrator can write his scheduling policy in a high-level language and then compile and link his policy into the scheduling module.

The second layer is an interpreter for a simple language, MIL, that allows the administrator to construct *filters* to control system behavior. Filters are logical predicates that

take as input two description vectors, and return a value reflecting the affinity of the two vectors. Both interface layers support administrative autonomy by allowing the administrator to customize the scheduling policy.

4.7 Example Execution of a Scheduling Module

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

A user on the local host submits a task to the distributed system, causing a **schedule request** event. The request includes a TDV describing the task. The scheduling module invokes a task filter, comparing the TDV with the SDV of each neighbor as well as the SDV for itself, and determines which SDV most closely matches the TDV according to the local policy. If the local system's SDV most closely matches, the local system accepts the task. If the SDV most closely matches a neighbor, the request is forwarded to the neighbor.

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

After a short time, a recalculation timeout event occurs, the scheduling module uses the data-collection functions to determine the local system state, and stores it in an SDV. The module then creates an update message to send to each neighbor, based on its own SDV and the SDVs it has received from its neighbors.

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

5 Support for Heterogeneity

Now that we have explained the function of the scheduling module and its support for autonomy in general, we will examine specific ways in which it supports task placement in heterogeneous distributed systems.

We will give a simple example in the MESSIAHS Interface Language (MIL) to demonstrate the support for heterogeneity. We will not attempt to describe MIL here; interested readers are referred to [3] or [4] for a detailed de-

```

begin combining
1.  string $out.proctype
    not match($out.proctype, "SPARC"):
        set $out.proctype + ":SPARC";

2.  string $out.OSname
    not match($out.OSname, "SunOS4.1"):
        set $out.OSname + ":SunOS4.1";

3.  string $out.proctype
    not match($out.proctype, $sys.proctype):
        set $out.proctype + $sys.proctype;

4.  string $out.OSname
    not match($out.OSname, $sys.OSname):
        set $out.OSname + $sys.OSname;
end
begin schedfilter
5.  $sys.address == $me.address and
    match($sys.proctype, $task.proctype)
    and match($sys.OSname, $task.OSname):
        max(200 - (100 * int($sys.loadave)),
            0);

6.  match($sys.proctype, $task.proctype)
    and match($sys.OSname, $task.OSname):
        max(400 - (100 * int($sys.loadave)),
            0);
end

```

Figure 4: partial remote execution specification

scription. Figure 4 shows part of a definition for a simple remote execution facility for a SPARC IPC, similar to that provided by Condor [1].

The combining rules in lines 1 and 2 ensure that the processor type variable, `proctype`, contains the string `:SPARC` and that the operating system variable `OSname` contains the string `:SunOS4.1`. The `proctype` and `OSname` variables are then inserted into the outgoing SDVs for this system. Lines 3 and 4 propagate incoming processor and operating system names to outgoing description vectors.

The example schedule request filter (lines 5 and 6) computes a rating function in the range [0, 200] for the local system, and [0, 400] for remote systems (remote execution is preferred). The scheduling request rules examine the processor type and operating system requirements of the

incoming TDV, matching them against known SDVs, and assign a priority to each match based on the system load average. The scheduling module automatically attempts to schedule the task on the system with the highest rating.

This module will only accept for execution those tasks that specify the SPARC architecture and SunOS4.1. If the TDV specified an SGI running IRIX 5.2, the scheduling module would not accept it for local execution, but would attempt to schedule the task on a neighboring system containing an SGI machine, if one exists. In this way, heterogeneous machines can cooperate in the same distributed system.

6 Conclusions and Future Work

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 showed how MESSIAHS supports heterogeneity, which is a special case of design 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.

There are several features of MESSIAHS that support autonomy:

- 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 and design autonomy,
- a request/reply paradigm supporting execution autonomy, and
- proxy acceptance to support communication autonomy.

The software currently runs on Sun-3 and SPARC workstations under SunOS 4.1. We are porting the software to IRIX, Linux, and VSTa. Preliminary performance results (see [4]) indicate a typical overhead of less than 10%

for dynamic algorithms, as compared to a perfect schedule generated off-line. We are quite pleased with these results, and will be conducting additional performance tests to determine the effects of data compaction on the scheduling algorithms.

Further information on the MESSIAHS mechanisms is available in [3, 4] and from the author.

References

- [1] A. Bricker, M. Litzkow, and M. Livny. Condor Technical Summary. Technical Report 1069, Department of Computer Science, University of Wisconsin-Madison, January 1992.
- [2] 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.
- [3] 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.
- [4] S. J. Chapin. Scheduling Support Mechanisms for Autonomous, Heterogeneous, Distributed Systems. Ph.D. Dissertation, Purdue University, 1993.
- [5] 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.
- [6] C. A. Gantz, R. D. Silverman, and S. J. Stuart. A Distributed Batching System for Parallel Processing. *Software—Practice and Experience*, 19, 1989.
- [7] 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.
- [8] S. D. Ostermann. Reliable Messaging Protocols. Ph.D. Dissertation, Purdue University, 1994.
- [9] XDR: External Data Representation Standard. Sun Microsystems Inc., June 1987. RFC 1014.