Constructing Distributed Schedulers Using the MESSIAHS Interface Language

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

Eugene H. Spafford
Dept. of Computer Sciences
Purdue University
West Lafayette, IN 47907–1398
spaf@cs.purdue.edu

Abstract

The MESSIAHS *project is investigating mechanisms that support task placement in heterogeneous, distributed, autonomous systems.* MESSIAHS *provides a substrate on which scheduling algorithms can be implemented. These mechanisms were designed to support diverse task placement and load balancing algorithms.*

As part of this work, we have constructed an interface layer to the underlying mechanisms. This includes the MESSIAHS *Interface Language (MIL) for constructing distributed schedulers. This paper gives an overview of* MESSIAHS*, describes a sample interface layer in detail, and gives example implementations of well-known algorithms from the literature built using this layer.*

1 Introduction

Recent initiatives in high-speed, heterogeneous computing have spurred renewed interest in large-scale distributed systems, and the desire for better utilization of existing resources has contributed to this movement. A typical departmental computing environment already has a substantial investment in computing equipment, including dozens or hundreds of workstations. Studies have shown that the utilization of this equipment can be as low as 30% of capacity [17, 23].

A solution to this problem is to conglomerate the 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. Large-scale distributed systems can have a combined computing power outperforming that of supercomputers [22].

A central element of effective utilization of such systems is task scheduling. Task scheduling has two components: macro-scheduling (also defined as global scheduling [8] and task assignment [24]) and micro-scheduling (or local scheduling [8]). Macro-scheduling chooses where to run a process, while micro-scheduling selects which eligible process to execute next on a particular processor. All further uses of the term *scheduling* in this paper refer to macro-scheduling.

The MESSIAHS[1] system [9, 10, 11] provides a set of mechanisms that facilitate scheduling in distributed, heterogeneous, autonomous systems. For our purposes, distributed, or loosely-coupled, systems communicate via message passing rather than a shared memory bus. Heterogeneous systems may have different instruction set architectures, data formats, and attached devices. All policy decisions in autonomous systems are made locally. Our vision of distributed systems includes all three attributes, connecting machines of different architectures with individual administrative authorities via a communications network. Section 2 gives precise definitions of autonomy and heterogeneity.

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 up to the national organization, has complete control over all the computers it may wish to use. An example of such usage is when two research organizations pool

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

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

their resources to solve a common problem.

Heterogeneity is important because it yields the most cost-effective and efficient method for performing some computations. 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. In other cases, tasks such as text processing or high-level language interpretation may be independent of any single architecture.

Our subsequent uses of the terms *distributed system* or *system* refer to a distributed, autonomous, heterogeneous system, and *node* refers to an individual machine within an autonomous system. Our definition of system includes a single machine, as well as two homogeneous workstations communicating via a local-area network. This definition also encompasses systems as complex as thousands of machines, including personal computers, workstations, file and computation servers, and supercomputers, spread among several remote sites and connected by a wide-area network.

Within this distributed system, each individual system has its own policy for deciding when to accept or remove tasks. The local administrator defines this policy, which is implemented over the MESSIAHS mechanisms via an interface layer. The interface layer provides a virtual machine interface; the underlying mechanism can be presented to algorithm writers in various ways. The language described here provides an interface that is easy to use, yet powerful enough to implement a wide variety of scheduling algorithms. Primitive operations are supplied to access system and task state information, manipulate tasks, and control the behavior of the local system.

This approach is distinct from that taken in distributed programming systems such as PVM [18] in which the program distribution is visible, and even forced upon, the programmer. The MESSIAHS approach more closely reflects that taken in Condor [7], which schedules processes invisibly for the programmer. Program distribution is under the control of the autonomous system, and therefore the administrators, rather than the programmer.

2 The MESSIAHS Architecture

MESSIAHS supports task placement in distributed systems with hierarchical structure based on administrative domains, modeled by directed acyclic graphs. 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[2]. The *neighbors* of a system are its children, parents, and siblings.

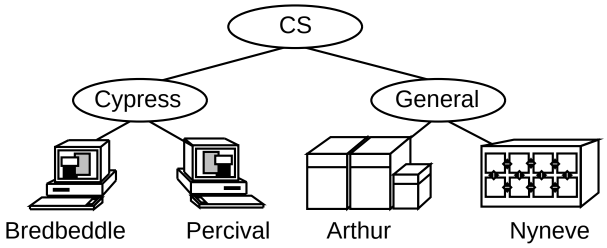

Figure 1: a sample distributed system

Figure 1 shows an example distributed system based on the Purdue University Computer Sciences department. In the example, the Computer Sciences department contains two administrative domains, Cypress and General. Cypress in turn encapsulates the research machines belonging to the Cypress project, and General contains the general purpose servers for the department. Bredbeddle and Percival are children of Cypress, 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.

Each module only exchanges status information 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, Arthur receives SDVs only from Nyneve and General, and sees no information that can be directly related to Percival or Bredbeddle. The capabilities of Bredbeddle and Percival are subsumed and combined within General's SDV.

Individual systems enjoy *execution autonomy*, *communication autonomy*, *design autonomy*, and *administrative autonomy* as defined in [11, 16, 14]. Execution autonomy means that each system decides whether it

[2]These correspond to the formal definitions of father, son, and brother found in [1].

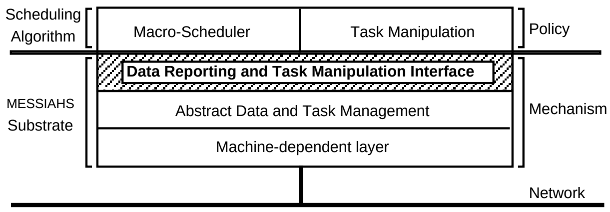

Figure 2: structure of a MESSIAHS scheduling module

will honor a request to execute a task; each system also has the right to revoke a task that it had previously accepted. Communication autonomy means that each system decides the content and frequency of state advertisements, and what other messages it sends. A system is not required to advertise all its capabilities, nor is it required to respond to messages from other systems. Design autonomy gives the architects of a system freedom to design and construct it without regard to existing systems, yielding heterogeneous systems.

Administrative autonomy means that each system can have its own usage policies and behavioral characteristics, independent of any others. In particular, a local system can run in a manner counterproductive to a global optimum. In the usual case, scheduling modules will cooperate, but administrators must be free to set their local policies or they will not participate in the distributed system. Both [7] and [17] note that users are willing to execute remote jobs on their workstations if the scheduling policy places higher priority on local jobs.

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 shaded layer, data reporting and task manipulation, is the focus of this paper. This layer presents the user with the interface to the MESSIAHS mechanisms. The administrator supplies the topmost layer, which embodies the scheduling policy for the system.

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.

3 Support for Scheduling Policies

As seen in figure 2, the scheduling policy is implemented over the interface layer. Through the interface layer, MESSIAHS either directly provides or supports five mechanisms that can be used to construct scheduling policies. These five mechanisms are system description, decision filters, task revocation, data combination and condensation, and node configuration and behavior customization.

3.1 Intrinsic Mechanisms

MESSIAHS uses a mechanism called *description vectors* to characterize available resources and requests for resources. A system description vector, or SDV, lists the capabilities of an autonomous system. A task description vector, or TDV, describes the resources required by a computational job. Description vectors contain a fixed portion that is optimized for task placement support, and an extensible portion that administrators can use to implement new scheduling policies or to extend the basic descriptions of requirements or abilities.

Table 1: fixed portion of a system description vector

field name	purpose
address	address of the system
module	id of module on this system
nsys	number of systems described by the vector
ntasks	total number of tasks currently accepted by the system
nactivetasks	number of active tasks running on the system
nsuspendedtasks	number of inactive tasks waiting on the system
willingness	desire of the system to take on new tasks
loadave	an estimate of the load average for the entire system
Procclass	information on the different classes of processors in the system

To determine the basis for the fixed portion of the description vector, we reviewed 18 algorithms from the existing scheduling literature [2–5, 7, 13, 17–20, 22–24, 26–28, 30, 31]. Table 1 depicts the resulting data set. We found that only two characteristics—processor speed and inter-processor communication time estimates—were used by more than four algorithms. Therefore, we included processor speed estimates in the description vector and provide a mechanism to determine inter-system communication time. We also augmented SDVs with other data items that we expect will be useful to writers of future scheduling algorithms. This data supports the common case, as represented by the surveyed algorithms, while the extension mechanism allows the inclusion of special-purpose data.

The `address` and `module` fields uniquely identify a scheduling module: the address specifies a machine, and the module indicates which module on that machine. MESSIAHS allows multiple modules to run on a single machine (see [9]). The `nsys` field indicates how many systems the vector represents; just as a distributed system encapsulates multiple subordinate systems, the description vector for a system contains information describing its component systems. The `ntasks`, `nactivetasks`, and `nsuspendedtasks` list the number of total tasks, running tasks, and suspended tasks for the system. The `willingness` gives the rough probability that the system will accept a new task, and `loadave` estimates the computational load on the entire system. The `Procclass` field is an array of records describing statistical measures of the processor utilization, processor speed, free memory, and disk space.

Execution autonomy mandates the ability to remove a task from execution on the local system. Aborting a running task fulfills the autonomy requirements, but does not support load-balancing algorithms based on process migration. Therefore, MESSIAHS includes mechanisms to kill, checkpoint, suspend, resume, and migrate jobs.

In support of administrative and communication autonomy, tunable parameters affect the general behavior of the node. These parameters are independent of any single scheduling policy, and effect all polices running on the node. These four parameters are listed in table 2.

The recalc_timeout field and revocation_timeout fields determine how often prescribed events occur. The SPECint92 and SPECfp92 are measures of processor speed using the SPEC benchmark suite [30]. The SPEC benchmark suite consists of applications-oriented programs, specifically selected to represent real-world workloads.

The machine architecture type (e.g. SPARC or VAX) does not appear as a universal parameter because many jobs are architecture independent. For example, text formatting requests require the presence of a particular text processing package, but do not depend on the underlying architecture.

Table 2: general state parameters

parameter	purpose
recalc_timeout	suggested period, in seconds, between recalculations of the local system description
revocation_timeout	period, in seconds, between checks for possible task revocation
SPECint92	the integer performance rating of the node, per the SPEC integer benchmark.
SPECfp92	the floating point performance rating of the node, per the SPEC floating point benchmark.

3.2 Supported Mechanisms

MESSIAHS supports the use of *filters* to implement scheduling policies. Decision filters take two description vectors as input, and return an integer value denoting how well the two vectors match according to the local policy. Larger values indicate closer matches. Scheduling modules employ filters to determine where to attempt scheduling a task (including on the local node), and what tasks are eligible for migration or revocation.

MESSIAHS allows multiple scheduling policies to operate within the system simultaneously, and a single node can support two or more scheduling policies. For example, batch queues for text processing, remote compilation, and remote program execution could all coexist within the same distributed system, each with its own individual scheduling policy. The administrator for each node could determine whether that node would participate as a server for any or all of the services.

Communication autonomy requires that the local policy control the flow of information out of a system. This mandates a mechanism to combine and compact the data set, and to allow the advertisement of restricted sets of information. In addition, data condensation is essential to avoid arbitrary limits on scaling the mechanisms. If systems concatenated all the data describing subordinate systems, the resources required to transmit and process a description vector would soon outstrip the capabilities of many networks and processors.

Unfortunately, some information loss is unavoidable if data compression takes place. Recall that in our example system, Arthur has no first-hand information about Bredbeddle or Percival. Therefore, Arthur might misdirect scheduling requests to General, based on the union of Percival's and Bredbeddle's abilities. For example, if Percival had 100 megabytes of free disk space and 4 megabytes of memory, while Bredbeddle had 10 megabytes of disk space and 32 megabytes of memory, the scheduling module on Arthur might mistakenly think that resources were available to execute a task requiring 16 megabytes of memory and 50 megabytes of disk space. These misdirected requests cause a small efficiency loss, but no tasks will be misscheduled as a result.

4 The Language

The shaded interface layer shown in figure 2 provides scheduling algorithms with access to lower-level mechanisms. We have chosen to provide a simple programming language as our interface, similar to that used in Univers [6]. Other interfaces are possible, such as a library of high-level language functions (see [9]).

The MESSIAHS Interface Language (MIL) contains direct support for dynamic scheduling algorithms, without precluding support for static algorithms. Static algorithms consider only the system topography, not the state, when calculating the mapping. Dynamic algorithms take the current system state as input, therefore the resultant mapping depends on the state (see [8]). Figure 3 depicts the structure of an MIL program. The grammars for deriving the various rules, along with explanations of their semantics, appear in the rest of this section.

4.1 Expressions and Types

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

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

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

Figure 3: MIL specification template

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

int-binop	$\rightarrow$	+ | − | / | * | mod | & | **|** | max | min
int-expr	$\rightarrow$	*int-expr int-binop int-expr* | (*int-expr*) | *integer* | int(*float-expr*) | *id*
string-expr	$\rightarrow$	*string-expr* + *string-expr* | (*string-expr*) | *string* | *id*

float-binop → + | − | / | * | max | min
float-expr → *float-expr float-binop float-expr* | (*float-expr*) | *float* | float(*int-expr*) | *id*

comp → < | > | = | >= | <= | <>
bool-binop → and | or | xor
bool-expr → *bool-expr bool-binop bool-expr* | not *bool-expr* | *int-expr comp int-expr* | *float-expr comp float-expr* | *string-expr comp string-expr* | match(*string-expr, string-expr*) | (*bool-expr*) | true | false | *id*

4.2 Access to Intrinsic Mechanisms

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

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

The node state section is a list of types, identifiers, and constant values. Node state declarations are parameters that affect system state. Unlike the extension variables, they do not directly appear in the system description vector. The four node state parameters are specint92, specfp92, recalc_timeout, and revocation_timeout. The specint92 and specfp92 parameters list the speed of the host in terms of the SPEC benchmarks [30]. The recalc_timeout and revocation_timeout parameters determine the timeout periods for the associated events.

4.3 Filters and Data Combination

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

filter-stmt → *bool-expr : int-expr ;*

A return value of `0` indicates that there is no match. A negative value indicates an error, and a positive value measures the affinity of the two vectors. As noted earlier, higher values indicate a better match. If the guard expression uses an undefined variable, the guard evaluates to `false`. If the integer expression references an undefined variable, the filter returns `-1`, indicating an error. With appropriate extension variables and guards, a single scheduling module can serve multiple scheduling policies as stated in section 3.2.

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

int-action → discard | set *int-expr*
float-action → discard | set *float-expr*
bool-action → discard | set *bool-expr*
string-action → discard | set *string-expr*

combining-rule → int *id bool-expr: int-action* ; | float *id bool-expr: float-action* ; | string *id bool-expr: string-action* ; | bool *id bool-expr: bool-action* ;

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

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

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

Figure 4: a simple MIL specification

4.4 Specification Evaluation

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

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

4.5 A Small Example

Figure 4 shows a simple MIL specification for a SPARC IPC participating in a distributed LaTeX text-processing system. Line 1 in the node state section sets the period for SDV recalculation at 60 seconds. Every minute, each participating system will compute its SDV and forward updates to its neighbors.

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

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

5 Example Algorithms

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

5.1 Distributed Batch

The MITRE distributed batch [17], Condor [7], and Remote Unix [23] systems support general-purpose distributed processing for machines running the UNIX operating system. Figure 5 lists a short specification file for a SPARC IPC participating in a distributed batching system. The state rules (lines 1–4) give the speed ratings for an IPC and the recalculation and revocation timeout periods.

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

The example schedule request filter (lines 9 and 10) computes a rating function in the range [0, 200] for the local system, and [0, 400] for remote systems. The scheduling request rules ensure that the processor type and operating system match, and assign a priority to a match based on the system load average. Because there is no provision for requesting tasks from a busy system, the section for task request rules is empty.

Hosts participating in the batch system preserve autonomy by varying the parameters of the schedule

```
  begin state
1.   float $SPECint92          13.8;
2.   float $SPECfp92           11.1;
3.   int   $recalc_period      30;
4.   int   $revocation_period  30;
  end
  begin combining
5.   string  $out.proctype
     not match($out.proctype, "SPARC"):
         set $out.proctype + ":SPARC";

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

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

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

10.  match($sys.proctype, $task.proctype)
     and match($sys.OSname, $task.OSname):
       max(400 - (100 * int($sys.loadave)),
           0);
  end
  begin revokefilter
11.  true:   1;
  end
  begin revokerules
12.  $me.loadave > 2.0 and
     $me.nactivetasks > 2:
        suspend;

13.  $me.loadave < 1.0 and
     $me.nsuspendedtasks > 0:
        wake;
  end
```

Figure 5: remote execution specification

request filter. For example, tasks submitted by a local user can be given higher priority by basing the rating function on the source address of the task.

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

5.2 Load Balancing

Several researchers have investigated load balancing and sharing policies for distributed systems, such as those described in [12], [15], and [26].

The *greedy load-sharing algorithm* [12], makes decisions based on a local optimum. When a user submits a task for execution, the receiving system attempts to place the task with a less busy neighbor, according to a weighting function. If no suitable neighbor is found, the task is accepted for local execution.

The suggested weighting function to determine if a task should be placed remotely is $f(n) = n \ div \ 3$, where n is the number of tasks currently executing on the local system. The algorithm searches for neighbors whose advertised load is less than or equal to one-third the local load. Because the greedy algorithm depends on local state, it is dynamic.

The policy specification in figure 6 implements a variant of the greedy algorithm. The original algorithm used a limited probing strategy to collect the set of candidates for task reception. The version in figure 6 sets the recalculation and retransmission periods low (line 1), and depends on the SDV dissemination mechanism to determine the candidate systems.

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

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

```
   begin state
1.   int $recalc_period 5;
   end
   begin combining
2.   int $out.minload
     ($sys.address == $me.address):
       set min($out.minload, $me.ntasks);

3.   int $out.minload true:
       set min($out.minload, $sys.minload);
   end
   begin schedfilter
4.   $sys.address == $me.address: 1;

5.   $sys.minload <= ($me.ntasks / 3):
        max(100 - $sys.minload, 2);
   end
```

Figure 6: specification for greedy load sharing

6 Concluding Remarks

The mechanisms provided by the MESSIAHS system and MIL support global task scheduling and load sharing in scalable distributed systems. These mechanisms also protect the autonomy of the individual systems, while uniting heterogeneous machines into a coherent distributed system.

The language presented here is simple and expressive. It addresses two neglected areas of distributed scheduling, heterogeneity and autonomy. MIL supports a broad range of existing scheduling algorithms, while enabling rapid development, prototyping, and analysis of new policies.

Because of its simplicity, MIL is somewhat limited. It cannot store history and has no control flow or looping constructs. Because of this, scheduling algorithms that accept multiple tasks and a set of system descriptions as input cannot be expressed precisely using this language. MIL also assumes that neighbors can be trusted to tell the truth in their SDV advertisements, and depends on a model of timely information exchange. A more complex approach that addresses these limitations, implemented as a set of library calls for ligh-level languages, appears in [9].

In summary, MESSIAHS embodies mechanisms supporting task placement in distributed, heterogeneous, autonomous systems. This support includes extensible mechanisms for implementing the local scheduling policy. This paper briefly described the MESSIAHS scheduling support mechanisms, defined a simple language for constructing schedulers, and gave sample implementations of representative scheduling policies using the language.

References

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

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

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

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

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

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

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

[8] 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.

[9] S. J. Chapin. Scheduling Support Mechanisms for Autonomous, Heterogeneous, Distributed Systems. Ph.D. Dissertation, Purdue University, 1993.

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

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

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

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

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

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

[16] 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.

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

[18] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM 3.0 User's Guide and Reference Manual*. Oak Ridge National Laboratory, Oak Ridge, TN 37831, February 1993.

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

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

[21] C. C. Hsu, S. D. Wang, and T. S. Kuo. Minimization of task turnaround time for distributed systems. In *Proceedings of the 13th annual international computer software and Applications Conference*, 1989.

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

[23] M. 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.

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

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

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

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

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

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

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

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

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