Dissemination of State Information in Distributed Autonomous Systems

Steve J. Chapin*
Dept. Computer Science
Thornton Hall
University of Virginia
Charlottesville, VA 22903
chapin@virginia.edu

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

Abstract

Many researchers are devising algorithms for task placement in distributed systems, but few are designing the necessary mechanisms to provide the information required by those algorithms. Fundamental to these mechanisms is an accurate means for information exchange between distributed systems.

The MESSIAHS project investigated the construction of a set of mechanisms to support task placement in autonomous, heterogeneous, distributed systems. In this paper we describe the semantics of the protocols used to exchange system state information within MESSIAHS, and develop formal models to prove that the protocols accurately propagate system description information throughout the system.

Keywords: parallel and distributed systems, task scheduling, autonomy, state dissemination

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

1 Introduction

Large collections of computing resources typically exist in commercial, academic, and governmental installations. Even greater resources are available when metasystems are constructed from multiple organizations' computing equipment. Metasystems combine heterogeneous computing resources from multiple administrative domains, and may scale to many thousands of hosts and millions of users. However, the organizations comprising a metasystem are likely to have different policies and goals for the use of their equipment which would normally preclude their shared use. Indeed, within organizations there are typically different administrative groupings of computing resources with conflicting policies.

To take full advantage of large-scale distributed computing, it is necessary to assign tasks to processors in such a way that these multiple, localized policies are accommodated.[1] Placement algorithms take a set of tasks and a description of the underlying multicomputer and devise an assignment of tasks to processors according to one or more optimizing criteria. Many researchers, including Sarkar [1], Lo [2, 3], Stone [4], Blake [5], Berman [6], and Weissman [7] have devised such algorithms.

However, most work in this area concentrates on policy decisions and assumes that mechanisms are available to support the proposed algorithms. We are investigating mechanisms to connect and support task placement in autonomous, heterogeneous, distributed systems. Examples of such support mechanisms are methods of describing tasks and processors, primitives to control the execution of tasks, and facilities to exchange state information among resource management modules within the distributed system. This last facility, information exchange, is necessary to provide the placement algorithms with accurate system descriptions as input.

To avoid overtaxing the underlying systems, and to ensure the scalability of the information exchange mechanisms, we have devised a technique of representing the state information for multiple systems in constant space [8]. One side effect of this process is that source information is lost—if the same state advertisement arrives at a site on two different paths, the receiver cannot recognize the messages as duplicates. In such an environment, If schedule writers are to have confidence that their schedulers function correctly, the underlying information dissemination mechanisms must perform properly. This means that the support mechanisms must not compromise the validity of advertised state information, either through overestimation or underestimation of available resources. In this paper, we define two aspects of proper state dissemination, called *completeness* and *correctness*, and develop a model for structuring dissemination patterns in distributed systems.

Using this model, we propose state dissemination rules for distributed systems with three different structures. We then prove that the rules are both complete and correct, and therefore will accurately provide data for scheduling algorithms. These dissemination rules have been implemented within the MESSIAHS[2] scheduling support system [8, 9], and we are applying the lessons learned as we construct resource management facilities for the Legion metacomputing system [10].

Section 2 gives a brief introduction to the MESSIAHS system and describes the hierarchical

[1]The process of task palcement is also called *scheduling*.

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

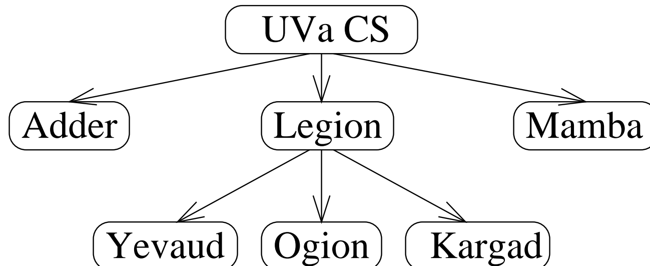

Figure 1: A subset of the machines in the UVa CS Dept.

structuring that forms the basis for the state dissemination rules. Section 3 lists definitions and notation that will be used to prove the integrity for the rules; the proofs appear in sections 4, 5, and 6. Section 7 describes some heuristics for choosing which of the dissemination rules should be used for a particular distributed system. Finally, section 8 gives our conclusions.

2 Background: the MESSIAHS System

We have stated that properly functioning scheduling support mechanisms will not alter state information as it flows through a distributed system. This change could be in the form of overestimation (e.g., including the capabilities of a single machine twice) or underestimation (e.g. failing to include a correctly reporting machine). To understand how such alterations could occur, we must first examine the MESSIAHS distributed system architecture.

Our target systems are structured in a hierarchical fashion based on administrative domains. A virtual system is composed of a set of subordinate virtual systems. Within each of these sets there can be many machines, which could be further grouped into virtual systems. At the lowest level, each machine is the sole member of a virtual system. We call an encapsulating virtual system a *parent*, and a subordinate system a *child*. Children with the same parent are called *siblings*.

We represent our systems with acyclic, directed graphs, with encapsulating virtual systems as interior nodes and machines as leaf nodes. Each node is the root of a graph representing a virtual system, and is itself a virtual system. In some cases, the virtual nodes map directly onto machines, and in other cases they do not. For example, figure 1 displays part of the administrative structure at the University of Virginia Department of Computer Science. Within the department, there are several generally accessible machines such as Adder and Mamba, as well as several research projects. One of these project is the Legion project, which has administrative authority over a set of machines including Ogion, Kargad, and Yevaud.

MESSIAHS attempts to sacrifice the least autonomy for participating systems. There are four types of autonomy in distributed systems, as defined in [11, 12, 13], and refined in [14]: *execution autonomy*, *communication autonomy*, *design autonomy*, and *administrative autonomy*. Execution autonomy allows each system to decide whether it 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 allows each system to decide the content and frequency of state advertisements, and any 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 allows each system to 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 experience shows they will not participate in the distributed system [15, 16].

In MESSIAHS, each virtual system in the hierarchy has a scheduling support module that is responsible for maintaining the set of information required by the scheduling policy. To accomplish this, the scheduling module communicates with neighboring scheduling modules in the system to exchange state information and processing requests. The scheduling module controls task placement for tasks passing into its subdomain–a user on Mamba running a task on Ogion must have the approval of both the Legion and Ogion scheduling modules. Our method for supporting scheduling decisions has three main parts: the *system description vector*, the *task description vector*, and the update protocol used to communicate between systems. System description vectors contain state description information, including system processing load, memory statistics, processing capabilities, and storage capacities. Task description vectors describe the resources required by a task. The update protocol sends system description vectors between modules.

The model for update flow is that a module collects several description vectors, adds information describing the local system, and condenses the resulting set of description vectors into one vector of constant size to facilitate scalability. This is achieved through a statistical representation of system capabilities, as seen in [8]. During this process, the system can alter the information content of the vector in one of three ways: the system can underestimate resources, overestimate resources, or change values within the description vectors that do not represent quantities.

Although allowing the underestimation of resources seems counterproductive at first inspection, it is necessary to support communication autonomy. For example, a research group might wish to make a group of workstations available to outside agencies for general purpose computation, but to allow only members of the research group access to a parallel processor. Communication autonomy allows the research group to prohibit advertisement of the parallel processor's resources outside the research group. Underestimation of resources will not cause tasks to be erroneously scheduled, but it might leave resources underutilized.

For example, Kargad is a multiprocessor workstation and is reserved for use by Legion group members. Yevaud and Ogion are available for external use. Thus, the Legion scheduling module would advertise Ogion's and Yevaud's capabilities to the outside world (UVa CS, Adder, and Mamba), but Legion would not advertise Kargad. Kargad might sit idle while Adder and Mamba are overloaded, but that is perfectly acceptable.

Therefore, the communication mechanisms assume that systems can and will underestimate resources in support of communication autonomy. However, systems are not expected to overestimate or change resource information. Policies can be written that violate these assumptions, but no assurances can be made about the performance of such policies. The key

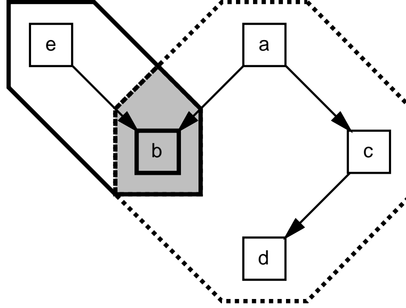

Figure 2: Global and local completeness

attribute we wish to show is the inverse of the well-known "garbage in, garbage out" principle: given correct description data, sound mechanisms will faithfully propagate that data throughout the distributed system. This is a classic separation of policy from mechanism—policies determine the input data, and the mechanisms disseminate that data. All further discussion of soundness in this paper focuses on the mechanisms.

3 Definitions and Notation

We define sound information dissemination mechanisms to be those that guarantee exactly once semantics—information advertised by a host reaches every other host within the system exactly once. There are two aspects of exactly once semantics: guaranteeing that advertised information reaches all other nodes within the system, defined here as *completeness*, and guaranteeing that information is not duplicated during advertisement, defined here as *correctness*. The correctness constraint arises from one of the basic assumptions for the information dissemination mechanisms: overestimation of system resources must be avoided.

We distinguish between two subtypes of completeness: *global completeness* and *local completeness*. Information advertised with a globally complete state dissemination mechanism reaches all other nodes within the entire distributed system, while a locally complete mechanism guarantees that advertised information reaches all other nodes with the same root. As an example distinguishing local completeness from global completeness, consider figure 2. The system rooted at a contains the nodes $\{a, b, c, d\}$, and the system rooted at e contains the nodes $\{b, e\}$. With a globally complete mechanism, information advertised by e will reach a, b, c, and d. With a locally complete mechanism, that information will only reach b. Similarly, with a globally complete mechanism, an advertisement by d reaches e, but not with a locally complete mechanism.

At first glance, a locally complete mechanism might seem less desirable than a globally complete mechanism. However, the administrative hierarchies that give rise to the system structure provide a motivation for local completeness. In terms of administrative domains, b is a shared resource jointly administered by e and a. By joining with e to administer b, a has not granted e the right to use a, c, or d. Local completeness allows administrators to automatically restrict data advertisements and scheduling requests to a rooted subgraph, provides autonomy support, and can form the basis for security mechanisms. For example,

an industry consortium might purchase collective computing resources for use by the group, but not want the individual members' resources to be visible to other members.

Our proofs use a multiset notation to denote the system description vectors passed between systems. A multiset is simply aset that may contain duplicate elements. Even though the actual data passed through the update vector is untagged, the model uses the name of a system to represent its capabilities in the multiset. For example, the set $\{a, b, c\}$ represents a description vector that contains the capabilities of the distributed system combining a, b, and c. This notation makes it obvious when a description vector violates the correctness constraint by including a node's capabilities multiple times, because the name of the node appears multiple times in the multiset.

We define four operators on multisets: $\uplus$, $\cap$, $\setminus$, and $\|\|$. The operator $\uplus$ is the multiset union operator with duplicate inclusion. For example, $\{a, b\} \uplus \{a, c\} = \{a, a, b, c\}$. The $\uplus$ operator models the coalescing of two system description vectors into one. The notation $\biguplus_{i \in range} \mathcal{S}_i$ represents the mapping of the $\uplus$ operator over multiple multisets, and ϕ denotes the empty multiset. $\setminus$ is the difference operator for multisets, and is defined to remove as many instances of an item from the first set as appear in the second set, e.g. $\{a, a, b, c\} \setminus \{a, b, d\} = \{a, c\}$. The intersection operator for multisets, $\cap$, yields a multiset containing the lesser number of each element common to two multisets, e.g. $\{a, b\} \cap \{b, b, c\} = \{b\}$. The $\|\|$ operator returns the cardinality of the multiset, e.g. $\|\{a, a, b\}\| = 3$.

The next three sections contain proofs of semantics for various distributed system architectures, including common cases such as tree-structured systems.

4 Tree-structured Systems

This section defines and proves properties of update semantics for tree-structured systems. We use standard definitions for graph notation, with a graph $\mathcal{G}$ being composed of a set of vertices $\mathcal{V}$ and a set of directed edges $\mathcal{E}$. An individual edge in the graph is denoted by listing its endpoints, e.g. an edge from vertex v_1 to vertex v_2 is written (v_1, v_2).

Tree-structured graphs are motivated by existing administrative domains. The typical administrative domain within an organization is tree-structured, for example, the departmental structure in figure 1. Thus, these rules optimize for the expected case.

4.1 Combining Rules

For a node x, equations 1 through 6 give definitions for the sets containing the x's parent (Pa_x), children (Ch_x), siblings (Si_x), ancestors (An_x), descendants (De_x), and the root (Ro_x) of the tree containing x.

$$Pa_x = \{p \mid (p, x) \in \mathcal{E}\} \tag{1}$$

$$Ch_x = \{c \mid (x, c) \in \mathcal{E}\} \tag{2}$$

$$Si_x = \{s \mid (\exists p \mid (p, x) \in \mathcal{E}, (p, s) \in \mathcal{E}, s \neq x)\} \tag{3}$$

$$An_x = \{a \mid a \in Pa_x \text{ or } (\exists p \mid p \in Pa_x, a \in An_p)\} \tag{4}$$

$$Ro_x = \{r \mid (r = x \text{ or } r \in An_x), Pa_r = \phi\} \tag{5}$$

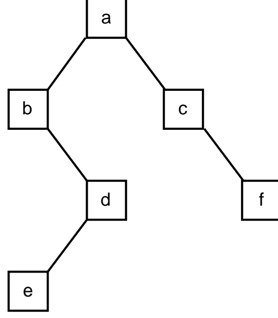

Figure 3: A tree-structured distributed system

Table 1: Example U and D sets

$U_a = \{a, b, c, d, e, f\}$	$D_a = \{a\}$
$U_b = \{b, d, e\}$	$D_b = \{a, b, c, f\}$
$U_c = \{c, f\}$	$D_c = \{a, b, c, d, e\}$
$U_d = \{d, e\}$	$D_d = \{a, b, c, d, f\}$
$U_e = \{e\}$	$D_e = \{a, b, c, d, e, f\}$
$U_f = \{f\}$	$D_f = \{a, b, c, d, e, f\}$

$$De_x \quad = \quad \{d \mid x \in An_d\} \tag{6}$$

Each system computes two sets of system status data to be passed to its neighbors. The U set is passed *up* to the system's parents and sideways to its siblings, and the D set is passed *down* to its children. The mechanisms may actually compute a separate set, S, for siblings, but in the case of tree-structures systems, this is the same as the set U. The notation U_x and D_x refers to the sets maintained by a system x within a tree-structured system. Equations 7 and 8 define these recursively in terms of the structure of the system. In practice, the total data stored for outgoing update vectors is independent of the number of children or siblings for a node, which helps to bound the resource usage of the mechanisms.

$$U_x = \{x\} \uplus \left(\biguplus_{i \,\in\, Ch_x} U_i \right) \tag{7}$$

$$D_x = \{x\} \uplus \left(\biguplus_{j \,\in\, Pa_x} D_j \right) \uplus \left(\biguplus_{k \,\in\, Si_x} U_k \right) \tag{8}$$

Informally, the U vectors include the description of the node and the U vectors from all its children. The D vector includes the description of a node, the D vector from its parent, and the U vectors from all of its siblings. The U and D vectors for the example system in figure 3 are in table 1. In terms of a subtree rooted at x, U_x describes the capabilities for all systems inside the subtree, and D_x describes the capabilities of all systems outside the subtree, plus the capabilities of x.

4.2 Proofs of Semantics

This section contains proofs that the mechanisms defined in equations 7 and 8 are globally complete[3] and correct. To prove that the semantics are complete, we must prove that a datum advertised by a node will reach all other nodes in the tree. To prove that the semantics are correct, we must prove that datum reaches other nodes at most once.

The following steps provide an outline of the proof:

1. prove that a system appears in a node's U vector if and only if the system is a member of the subtree rooted at the node,
2. prove that a system appears in a node's D vector if and only if that system is not a descendant of the node,
3. prove that no system appears more than once in a U vector, and
4. prove that no system appears more than once in a D vector.

Steps one and two prove completeness, and steps three and four prove correctness of the semantics for tree-structured systems. For tree-structured graphs, local and global completeness are the same.

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

> The U vector for a node represents the node and its descendants, and only the node and its descendants.
>
> Proof (by induction): Recall the definition of U_x in equation 7. First comes the proof of the $\Rightarrow$ implication.
>
> > For a leaf system, $U_x = \{x\}$, and the implication holds. This is the base case. For a non-leaf system, the induction step assumes that the implication is true for all children of x. Thus, $\forall c \in Ch_x$, U_c contains only descendants of c, and c itself. The second term of equation 7 adds only children of x and their descendants, which are also descendants of x. Therefore the second clause of the implication holds, and the $\Rightarrow$ case is true.
>
> Now for the $\Leftarrow$ implication:
>
> > By definition, x is always in U_x, so we need only prove that all descendants of x are members of U_x. At a leaf node there are no descendants, so the base case is proven. For the induction step, we assume that the implication holds for all children of x. Therefore, the second term in the definition of U_x adds all children of x, and all of their descendants, which is to say it adds all the descendants of x. Thus, $y \in De_x \Rightarrow y \in U_x$. $\square$

[3]For tree-structured graphs, local and global completeness are the same.

Lemma 2 $y \in D_x \Leftrightarrow y \notin De_x$

Any system that appears in D_x is not a descendant of x, and all non-descendants of x appear in D_x

First comes the proof of the $\Rightarrow$ implication.

Proof (by induction): At the root, $D_x = \{x\}$. By definition, $x \notin De_x$. For the induction step, assume that the lemma is true for the parent of the node x. Suppose that the lemma is false for x, i.e. $\exists y$ such that $y \in D_x$ and $y \in De_x$, and examine each of the three terms in the definition of D_x. y cannot be x, as $x \notin De_x$ by definition. y cannot come from the second term, as that would violate the induction assumption. From lemma 1 and the fact that the U vectors for siblings are disjoint because of the tree-structure of the graph, y cannot come from the third term. Therefore, y cannot exist, and the lemma is true for x in the inductive step. $\square$

Now comes the proof of the $\Leftarrow$ implication.

Proof (by induction): At the root, $D_x = \{x\}$. Because x is the root, all other nodes are descendants of x, so the base case is true. For the induction step, we partition the set of non-descendants of x into two groups: those that are not descendants of x's parent, p, and those that are descendants of p. The induction step assumes the implication holds for p.

All non-descendants of p are also non-descendants of x, and because of the induction assumption, are included in D_x by the second term in the definition of D_x. This leaves the descendants of p for consideration. From the $\Leftarrow$ implication of lemma 1, all siblings of x and their descendants are included by the third term in D_x. By definition, $x \in D_x$. Therefore, all descendants of p that are not descendants of x appear in D_x.

Therefore, all non-descendants of x appear in D_x. $\square$

Theorem 1 *The Completeness Theorem for Tree-Structured systems: Information describing each node reaches every other node within the system, using rules 7 and 8.*

Proof: By lemma 2, information describing all non-descendants of a node x appears at x. By lemma 1, information describing all descendants of x is visible to x. As x knows about itself, x receives a description of all nodes in the system, and the semantics are (globally and locally) complete. $\square$

The next two lemmas are crucial in establishing correctness. These lemmas state that U and D vectors do not contain duplicate entries.

Lemma 3 $\forall y \in U_x, y \notin (U_x \setminus \{y\})$
(No system is represented more than once in a U vector.)

> Proof (by induction): At a leaf, $U_x = \{x\}$, and the lemma is true. Assume that the lemma is true for all children of a node x.
>
> If the lemma is false, then $\exists y \mid y \in (U_x \setminus \{y\})$. Based on equation 7, either $y = x$ and $x \in U_c$ for some child c of x, y appears in the U vectors of multiple children of x, or a child of x has duplicates in its U vector. From lemma 1, $x \in U_c \Rightarrow x \in De_x$, which cannot be in an acyclic graph. Likewise, the tree-structure of the graph means that the U vectors for two children of a node are disjoint, so y cannot appear in the U vectors of multiple children of x. The induction assumption precludes the third possibility. Therefore, y cannot exist, and the lemma is proven. □

Lemma 4 $\forall y \in D_x, y \notin (D_x \setminus \{y\})$
(No system is represented more than once in a D vector)

> Proof (by induction): At the root, $D_x = \{x\}$, and the lemma is true. For the induction assumption, assume that the lemma is true for the parent of a node.
>
> If the lemma were false, then $\exists y \mid y \in (D_x \setminus \{y\})$. Based on equation 8, one of the following must be true:
>
> 1. $x \in D_p$ for $p \in Pa_x$ (interaction of the first and second terms)
> 2. $x \in U_k$ for some $k \in Si_x$ (interaction of the first and third term)
> 3. $D_p \cap U_k \neq \phi$ for $p \in Pa_x$ and some $k \in Si_x$ (interaction between the second and third terms)
> 4. $U_j \cap U_k \neq \phi$ for some $j, k \in Si_x, j \neq k$ (interaction between two members of the third term)
>
> Lemma 3 and the induction assumption eliminate the possibility of duplicates within a single incoming vector. By lemma 2, (1) cannot be true. For x to appear in a sibling's U vector, x would must be a descendant of its sibling, which violates the tree structure of the graph; this eliminates case (2). For (3) to hold, a system simultaneously be a descendant of p (corollary of lemma 1 and the fact that $k \in Ch_p$) and also not be a descendant (lemma 2). Therefore, (3) cannot hold. The disjointedness of U vectors for siblings eliminates case (4).
>
> Thus, y cannot exist and the lemma is proven. □

Theorem 2 *The Correctness Theorem for Tree-Structured Systems:*
No system's attributes appear in any update vector more than once, using rules 7 and 8.

> Proof: Lemma 3 states that no system appears more than once in a U vector. By Lemma 4, no D vector has duplicate entries. From lemmas 1 and 2, the D and U vectors arriving at a node are disjoint, and they do not contain the node itself. Therefore, the semantics for combining update vectors will not overestimate system resources, and are thus correct. □

In theorems 1 and 2, we have shown that an update protocol based upon the combining rules presented here will accurately disseminate system description information through a tree-structured distributed system.

5 A Subclass of Non-tree-structured DAG

Equations 7 and 8 cannot be applied to more general directed acyclic graphs—if a node has two parents with a common ancestor, the capabilities of the system will be overestimated (see the example in figure 4). Such organizational structures appear in practice when two research projects, e.g. b and c, pool funds to purchase a single machine, d. Both research projects have the machine within their administrative hierarchy. The resulting erroneous U and D sets are in table 2. U_a and D_d violate the correctness constraint because they overstates resources—the two paths between a and d cause this.

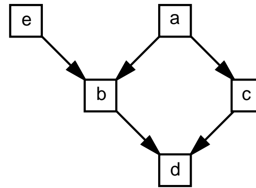

Figure 4: A non-tree-structured system

Table 2: Erroneous U and D sets for figure 4.3

$U_a = \{a, b, c, d, d\}$	$D_a = \{a\}$
$U_b = \{b, d\}$	$D_b = \{a, b, c, d, e\}$
$U_c = \{c, d\}$	$D_c = \{a, b, c, d\}$
$U_d = \{d\}$	$D_d = \{a, a, b, b, c, c, d, d, d, e\}$
$U_e = \{b, d, e\}$	$D_e = \{e\}$

5.1 A Refinement: Primary Parents

Introducing the notion of *primary parents* for each system and modifying the U and D vector definitions solve this problem. First, we select a subset of the edges in the graph that would form a spanning tree if they were undirected. Second, we designate the links that form the spanning tree as *primary links*. A path composed of primary links is a *primary path*. A primary link joins a *primary parent* to a *primary child*.

The notions of primary ancestor, primary descendant, and primary sibling are analogous to the notions of ancestor, descendant, and sibling defined earlier. Equations 9 through 13 define the sets of primary parents, children, siblings, ancestors, descendants, and roots for a node.

$$PP_x = \{p \mid (p, x) \in \mathcal{E}, \text{ and } (p, x) \text{ is a primary link}\} \quad (9)$$

$$PC_x = \{c \mid (x, c) \in \mathcal{E}, \text{ and } (x, c) \text{ is a primary link}\} \quad (10)$$

$$PS_x = \{s \mid (\exists p \mid p \in PP_s, p \in PP_x, s \neq x)\} \quad (11)$$
$$PA_x = \{a \mid a \in PP_x \text{ or } (\exists p \mid p \in PP_x, a \in PA_p)\} \quad (12)$$
$$PD_x = \{d \mid x \in PA_d\} \quad (13)$$
$$PR_x = \{r \mid (r = x \text{ or } r \in PA_x), PP_r = \phi\} \quad (14)$$

The primary links must have the following properties:

1. (no common ancestor) A node may have more than one primary parent, but no two primary parents of a node may share a common ancestor.
2. (preservation of roots) For any node x, if a root $R \in Ro_x$ in the original graph, then $R \in PR_x$ in the resulting graph.

Section 5.3 describes a method for choosing primary links that meet these criteria.

For example, in figure 4, b has two parents, a and e, which do not have a common ancestor, so both a and e may be primary parents of b. In contrast, both of d's parents, b and c, have a common ancestor in a, and therefore they cannot both be primary parents for d.

The proofs and semantics presented here assume that a proper set of primary links has already been selected for the graph. As a result, a primary path exists from each node to all of its roots, and there is exactly one primary path between any two nodes within the system. This does not reduce directly to the tree-structured case, because the links in the primary path do not necessarily form a directed spanning tree. Also, although it is not propagated, information still flows across non-primary links.

The combining rules for primary parents resemble those for tree-structured systems, with the following exceptions: only primary parents and primary siblings of a node incorporate U updates from that node; similarly, only primary children of a node incorporate its D updates. Other parents, children, and siblings receive the updates and can use them to make scheduling decisions, but do not include them in the computation of their own update vectors. Equations 15 and 16 incorporate the primary parent into the definitions of the U and D vectors. As in tree-structured systems, the data storage requirements for outgoing vectors are constant because.

$$U_x = \{x\} \uplus \left(\biguplus_{i \in PC_x} U_i \right) \quad (15)$$

$$D_x = \{x\} \uplus \left(\biguplus_{j \in PP_x} D_j \right) \uplus \left(\biguplus_{k \in PS_x} U_k \right) \quad (16)$$

In figure 5, the solid arrows represent the primary links, and the dashed line is a non-primary link between c and d. The U and D vectors for figure 5 are in table 3.

5.2 Proofs of Semantics for Primary Parents

This section proves that the semantics defined in equations 15 and 16 are correct and locally complete for DAG-structured systems that fulfill the constraints outlined previously.

Once again, there are four steps to the proofs of correctness and completeness:

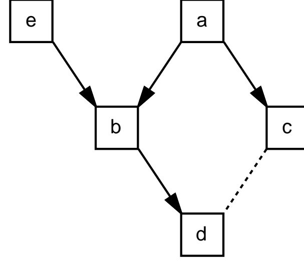

Figure 5: A non-tree-structured system with primary parents

Table 3: U and D sets

$U_a = \{a, b, c, d\}$	$D_a = \{a\}$
$U_b = \{b, d\}$	$D_b = \{a, b, c, e\}$
$U_c = \{c\}$	$D_c = \{a, b, c, d\}$
$U_d = \{d\}$	$D_d = \{a, b, c, d, e\}$
$U_e = \{b, d, e\}$	$D_e = \{e\}$

1. prove that a system appears in a node's U vector if and only if the system is a member of the primary subtree rooted at the node,

2. prove that a system appears in a node's D vector if and only if that system is not a primary descendant of the node, but shares a common primary root with the node,

3. prove that no system appears more than once in a U vector, and

4. prove that no system appears more than once in a D vector.

The first lemma states that the U vector for a node represents only the node itself, and all the node's primary descendants. The second lemma says that a host is represented in a node's D vector if and only if

1. the host is not a primary descendant of the node and

2. the host and the node share a common primary root.[4]

Lemma 5 $(y \in U_x) \Leftrightarrow (y = x \quad \text{or} \quad y \in PD_x)$
(A system appears in the U vector for a node if and only if the system is a primary descendant of the node, or the system and the node are the same.)

> The proof follows the same form as that for lemma 1, considering primary descendants instead of descendants.

[4] In the case of tree-structured graphs, this condition was met by all nodes because there was a single root. In the more general case, there may be multiple roots.

Lemma 6 $y \in D_x \Leftrightarrow (y \notin PD_x)$ *and* $(PR_x \cap PR_y \neq \phi)$
(A system appears in the D vector for a node if and only if the system is not a primary descendant of the node, and the system and the node share a primary root.)

Proof (by induction):

$\Rightarrow$ implication:

At the root, $D_x = \{x\}$. By definition, x is not a primary descendant of itself. Also by definition, x is its own primary root, so the lemma holds in the base case.

For the induction step, assume that the lemma is true for all primary parents of the node x. Choose a node $y \in D_x$, and examine the three terms of D_x from which y could arise. If $y = x$, the implication is true from the definition of PD_x. If y comes from the second term in the definition of D_x, then the induction assumption applies. If y comes from the third term, then from lemma 5, y is not a primary descendant of x. Because any two nodes that share a primary ancestor share a primary root, $PR_y \cap PR_x \neq \phi$. Therefore, the $\Rightarrow$ implication holds.

$\Leftarrow$ implication:

At the root x, the only node not a primary descendant of x that shares a primary root with x is x itself, and $x \in D_x$. This is the base case.

For the induction step, assume the implication is true for all parents of x. From the assumption, all nodes with the same roots as x that are not descendants of x's primary parents are included in D_x. From the definition of D_x, lemma 5, and the spanning tree imposed by the primary links, all primary siblings of x and their primary descendants appear in D_x. Therefore, the $\Leftarrow$ implication is true.□

Theorem 3 *The Local Completeness Theorem for Primary Parents: Information describing each node reaches every other node with the same primary root, under rules 15 and 16.*

Proof: By lemma 6, information describing all non-descendants of a node x, that share a primary root with x, appears at x. By lemma 5, information describing all descendants of x is visible to x. Because x knows about itself, x receives a description of all nodes in the system with the same primary root. □

The proof of correctness for primary parents uses two lemmas showing that neither U nor D vectors have duplicate entries.

Lemma 7 $\forall y \in U_x, y \notin (U_x \setminus \{y\})$
(No system is represented more than once in a U vector.)

This proof follows the same form as the proof for lemma 2, considering primary children instead of children.

Lemma 8 $\forall y \in D_x, y \notin (D_x \setminus \{y\})$
(No system is represented more than once in a D vector)

> Assume $\exists y | y \in (D_x \setminus \{y\})$. Then, based on equation 16, one of the following must be true:
>
> 1. $x \in D_p$ for some $p \in PP_x$ (interaction of the first and second terms)
> 2. $x \in U_k$ for some $k \in PS_x$ (interaction of the first and third term)
> 3. $D_m \cap U_k \neq \phi$ for $m \in PP_x$ and some $k \in PS_x$ (interaction between the second and third terms)
> 4. $D_j \cap D_k \neq \phi$ for any $j, k \in PP_x, j \neq k$ (interaction between two members of the second term)
> 5. $U_j \cap U_k \neq \phi$ for any $j, k \in PS_x, j \neq k$ (interaction between two members of the third term)
>
> By lemma 6, (1) cannot be true. Lemma 5 and the imposition of a spanning tree by the primary parent links eliminate cases (2) and (5). For (3) to hold, a system must simultaneously be a descendant of p (corollary of lemma 5 and the fact that $k \in PC_p$) and also not be a descendant (lemma 6). Therefore, (3) cannot hold. If (4) were true, then the two primary parents j and k must share a common root (lemma 6), and thus could not both be primary parents of x because of the spanning tree imposed by the primary links.
>
> Thus, a contradiction is reached and y cannot exist, so the lemma is proven. □

Theorem 4 *The Correctness Theorem for Primary Parents: No system's attributes appear in any update vector more than once using rules 15 and 16.*

> Proof: Lemma 7 states that no system appears more than once in a U vector. By lemma 8, no D vector has duplicate entries. From lemmas 5 and 6, the vectors arriving at a node are disjoint. Therefore, the semantics for combining update vectors will not overestimate system resources, and are thus correct. □

Theorems 3 and 4 show that the update protocol presented here will correctly and with local completeness disseminate system description information through a DAG-structured distributed system that meets the constraints detailed in the next section.

5.3 Constraints on Primary Parents

The proofs in the previous section assume that a spanning tree has already been imposed on the graph representing the system. However, this may not always be possible to do and still retain local completeness. This section defines a subclass of directed acyclic graphs for which spanning trees can be imposed without loss of local completeness. We call a spanning tree that preserves local completeness a *viable spanning tree*, and a graph with a viable spanning tree a *viable graph*.

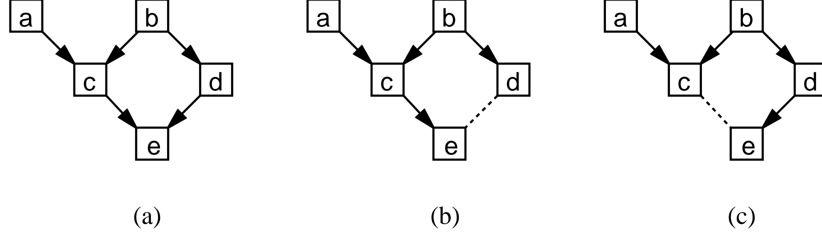

Figure 6: A distributed system and two spanning trees

We first examine situations in which the primary parent mechanism fails. For example, consider figure 6. The graph in part (a) represents a distributed system with multiple spanning trees. Sub-figures (b) and (c) represent the two choices for e's primary parent. Choosing c as the primary parent, represented in sub-figure (b), provides local completeness, because each node in the spanning tree is a primary descendent of all roots of which it was a descendant in the original graph. In sub-figure (c), e is not a primary descendant of a, so information describing e will not reach a. Therefore, only (b) represents a viable spanning tree for this system.

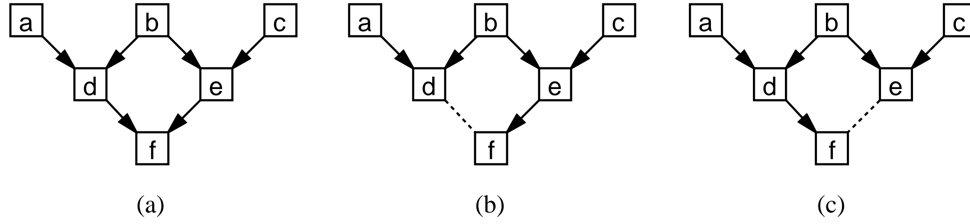

Figure 7: A distributed system with no viable primary parents

The graph depicted in figure 7 part (a) has no spanning tree that allows for local completeness. If f makes e its primary parent (sub-figure (b)), then a will not receive a description of f. If f makes d its primary parent (sub-figure (c)), then c will not receive information describing f. There is no viable spanning tree for this graph.

A key observation is this: for a spanning tree to be viable, and to therefore preserve completeness, it is necessary that a directed path from each of a node's roots to the node still exist in the subgraph covered by the spanning tree. The following predicate defines the viability property of a directed graph.

$$viable(\mathcal{V}, \mathcal{E}) \Leftrightarrow \forall v \in \mathcal{V}, \text{ there exists a subset } \mathcal{P}_v \text{ of } Pa_v \text{ such that } \left(\bigcup_{p \in \mathcal{P}_v} Ro_p\right) = Ro_v;\ \forall p, q \in \mathcal{P}_v \quad Ro_p \bigcap Ro_q = \phi \text{ and the primary links resulting from making all } p \in \mathcal{P}_v \text{ primary parents of } v \text{ form a spanning tree for the graph.} \tag{17}$$

Recall that Ro_x is the set of x's ancestors in the original graph that have no parents. Thus, the first part of this definition is a statement of viability in terms finding an exact

set cover of a node's root set by its parents' root sets. This is known to be an NP-complete problem (see [17]).

---

check_viable($\mathcal{R}$, $\mathcal{C}$, $\mathcal{P}$, Ro, i, lim)

1. if ($\mathcal{C} = Ro$) return $\mathcal{P}$;
2. for j in i . . . lim {
3. if ($\mathcal{C} \cap R_j = \phi$) {
4. $\mathcal{P}' \leftarrow$ check_viable($\mathcal{R}$, $\mathcal{C} \cup R_j$, $\mathcal{P} \cup \{j\}$, Ro, j+1, n);
5. if ($\mathcal{P}' \neq \phi$) return $\mathcal{P}'$;
6. }
7. }
8. return ϕ;

Figure 8: An algorithm to compute set covering for primary parents

Figure 8 contains a straightforward $O(2^n)$ algorithm, where n is the number of parents of a node, to test all possible combinations of the parent sets for coverage. The number of parents for a node is expected to be small, so this exponential growth is acceptable. This algorithm assumes that the root sets of the parents are stored in the set $\mathcal{R} = \{R_1, R_2, \ldots, R_n\}$. The algorithm takes as input the set $\mathcal{R}$, a cover set $\mathcal{C}$, a set of primary parents $\mathcal{P}$, a root set to be covered Ro, and two integers to act as counters and limit variables. It returns the list of parents that create the cover set, if an exact cover is possible. A return value of ϕ indicates failure.

Each node can independently determine if an exact cover set of its root set by its parents' sets exists with a call

$$\text{check_viable}(\textstyle\bigcup_{p \in Pa_x} \{Ro_p\},\ \phi,\ \phi,\ Ro_x,\ 1,\ \|Pa_x\|).$$

This can be done in parallel, and the parallel running time is $O(2^m)$, where $m = max(\|Pa_v\|), \forall v \in \mathcal{V}$. In practice, nodes have only a few parents (typically one or two), so the exponential running time of the algorithm is not an issue.

The set covering is a necessary condition for the graphs; it is not a sufficient condition to prove viability, because graphs exist which have viable parent coverings, but whose primary links do not form a spanning tree. For example, figure 9 has a viable parent set covering, but the primary links for the covering include all the edges in the graph, and do not form a spanning tree. f recieves a root set of $\{a\}$ from c, and a root set of $\{b\}$ from e. These form a viable set covering for all of f's roots, so the graph is viable. However, d will receive two copies of f's U vector, and vice versa, because of the two primary paths between f and d ($fcad$ and $febd$).

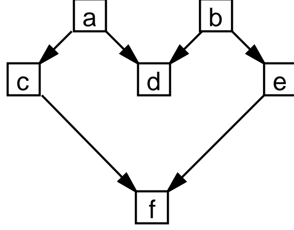

Figure 9: A graph with a viable set covering with duplicate paths

To verify that the primary links associated with the resultant primary parent set form a spanning tree for the graph, it is sufficient to show that there is unique path between nodes. This can be accomplished through a variation on reverse-path flooding. The node initiating the check broadcasts a status message with a unique token across all primary links. Nodes that receive the message broadcast it out over all primary links except the one over which the message was received. If a node receives the message twice, the primary links do not form a spanning tree, and the node can send an error reply to the originator of the message. This check will typically be made by a host attempting to join a distributed system.

6 General Directed Acyclic Graphs

Prior sections have described combination rules for certain subclasses of directed acyclic graphs. This section gives globally complete and correct combination rules for general DAG, with the tradeoff of higher computational and communication overhead.

6.1 General Combination Rules

First, a spanning tree is imposed on the graph using primary parents. The viability constraint of the previous section does not apply; any spanning tree will suffice. Unlike the previous semantics, these rules do not pass updates to siblings. Also, instead of sending the same vector to multiple neighbors, these rules require a distinct update vector for each parent or child. Thus, the storage and computational requirements are $O(\|PP_x\| + \|PC_x\|)$ for a node x. The rules for the update vector U_{xn}, passed from x to a neighbor n, are in equation 18. Note that the name U does not indicate a direction of transfer in this case.

$$U_{xn} = \{x\} \uplus \left(\biguplus_{p \,\in\, PP_x, p \neq n} U_{px} \right) \uplus \left(\biguplus_{c \,\in\, PC_x, c \neq n} U_{cx} \right) \tag{18}$$

The effect of this rule can be summarized as follows:

> Over a link from x to y, send out information describing x and the sum of information received by x on all other links.

Equation 18 employs the technique of a *split horizon update* found in network routing protocols [18, 19].

6.2 Proofs of General Rules

This section proves rule 18 globally complete and correct, given that a spanning tree has been imposed on the graph.

Theorem 5 *The Completeness Theorem: Information describing a node reaches all other nodes in the system, under rule 18.*

> Proof: the first term of rule 18 advertises a node to all its neighbors. The second and third terms of this rule propagate the information from one link to all other links. Therefore, each system is advertised to its neighbors, and because of the imposed spanning tree, the advertised information is propagated to all other nodes within the system. □

Theorem 6 *The Correctness Theorem: No system's attributes appear in any update vector more than once using rule 18.*

> Proof: Assume that there exists some y that appears more than once in an update vector for a node x. Under rule 18, only at a node y does the description information for y enter the system; all other nodes only propagate the information. Therefore, either $y = x$ and x appears in one of the incoming update vectors, or y appears in multiple incoming update vectors. The first indicates a cycle in the update vectors, and the second indicates two paths between x and y. Neither of these can occur in the presence of a spanning tree for a directed acyclic graph. Therefore, y cannot appear twice, and the theorem is proved. □

Theorems 5 and 6 demonstrate that the general semantics shown here are correct and globally complete.

7 Structuring Heuristics and Implications

Given a collection of machines that will use the MESSIAHS mechanisms to support distributed computation, the question arises of how to best impose a distributed system structure using the rules defined in this chapter.

The tree-structured rules are the simplest, require the least overhead, and apply to a majority of existing administrative domains where organizations do not collaborate to manage a common resource pool. The typical case for a small department, in which all machines are within a single domain, can be accommodated by making all the machines children of single virtual node. A department of moderate size might have an umbrella domain for the department as a whole, which encapsulates virtual systems for individual research projects, such as in our original CS Department example in figure 1.

The primary-parent rules are appropriate for administering shared resources, if a viable spanning tree can be determined. If no viable spanning tree can be determined, the general rules will provide global completeness for any administrative structure.

8 Conclusions

We are developing mechanisms to support scheduling in large distributed systems composed of multiple autonomous administrative domains. It is necessary to provide sound mechanisms to support the exchange of state information between machines in separate administrative domains.

We have described a formal model for system update dissemination in distributed, hierarchical, autonomous systems. The model defines the *completeness* and *correctness* properties for information dissemination in distributed systems, and uses a multiset-based notation to represent description vectors in such systems.

This paper described three sets of rules for state dissemination in distributed systems using the MESSIAHS mechanisms. The first model is correct and globally complete for systems structured as trees. The second model is correct and locally complete for systems structured on a subclass of directed acyclic graphs in which nodes have at most one common ancestor. The third and most general model is correct and globally complete for systems structured as directed acyclic graphs.

These dissemination rules form the basis for the update protocol in the MESSIAHS distributed scheduling support system. Because this system is based on provably sound dissemination rules, authors of scheduling algorithms who use the MESSIAHS mechanisms can be confident that the underlying mechanism will faithfully implement their policies.

References

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

[2] V. M. Lo. Task Assignment to Minimize Completion Time. In *Distributed Computing Systems*, pages 329–336. IEEE, 1985.

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

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

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

[6] F. Berman and R. Wolski. Scheduling from the perspective of the application. In *Proceedings of the 5th International Symposium on High-Performance Distributed Computing (HPDC-5)*. IEEE, August 1996.

[7] J. Weissman and X. Zhao. Scheduling parallel applications in distributed networks. *Journal of Cluster Computing*, to appear.

[8] S. Chapin and E. Spafford. Support for Implementing Scheduling Algorithms Using MESSIAHS. *Scientific Programming*, 3:325–340, 1994. special issue on Operating System Support for Massively Parallel Computer Architectures.

[9] S. J. Chapin. Distributed Scheduling Support in the Presence of Autonomy. In *Proceedings of the 4th Heterogeneous Computing Workshop, IPPS*, pages 22–29, April 1995. Santa Barbara, CA.

[10] A. S. Grimshaw, Wm. A. Wulf, and the Legion Team. The legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1), January 1997.

[11] 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.

[12] 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.

[13] 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.

[14] S. J. Chapin. Scheduling Support Mechanisms for Autonomous, Heterogeneous, Distributed Systems. Ph.D. Dissertation, Purdue University (Purdue CS TR-93-087), 1993.

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

[16] C. A. Gantz, R. D. Silverman, and S. J. Stuart. A Distributed Batching System for Parallel Processing. *Software–Practice and Experience*, 19, 1989.

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

[18] C. Hedrick. Routing Information Protocol. RFC 1058, Network Information Center, June 1988.

[19] G. Malkin. RIP Version 2–Carrying Additional Information. RFC 1388, Network Information Center, January 1993.