Distributed Top-Down Hierarchy Construction

David G. Thaler and Chinya V. Ravishankar
Electrical Engineering and Computer Science Department
The University of Michigan, Ann Arbor, Michigan 48109-2122
thalerd@eecs.umich.edu ravi@eecs.umich.edu

ABSTRACT: Hierarchies provide scalability in large networks and are integral to many widely-used protocols and applications. Previous approaches to constructing hierarchies have typically either assumed static hierarchy configuration, or have used bottom-up construction methods. We describe how to construct hierarchies in a *top-down* fashion, and show that our method is much more efficient than bottom-up methods. We also show that top-down hierarchy construction is a better choice when administrative policy constraints are imposed on hierarchy formation.

1 Introduction

Hierarchies are commonly used to achieve scalability in network protocols. Current protocols using multi-level hierarchies, such as DNS and X.500, typically require manual hierarchy configuration, an approach with considerable administrative overhead. We consider the problem of automating distributed hierarchy construction.

Generating a hierarchy is closely related to the problem of identifying *clusters* in data, since siblings in a hierarchy are related, and will be close to each other under an appropriate metric. Though algorithms exist for clustering nodes according to topological placement (e.g., [1–4]), they are often inadequate for inter-domain hierarchy construction in the Internet. Hierarchy construction in the Internet is complicated by the existence of administrative policy constraints. Centralized methods are not generally applicable in this domain, and to our knowledge, relatively little work has been done on *distributed* clustering methods.

There are two basic approaches to clustering: *agglomerative* (bottom-up) and *divisive* (top-down). In an agglomerative approach, each node begins as its own cluster; sets of clusters are then combined into larger clusters until the top-level cluster contains all nodes. In a divisive approach, all nodes begin in the same cluster, which is successively divided into smaller clusters until 1-node clusters are reached.

We can apply the same concepts to hierarchy construction. In a bottom-up scheme, all agents begin as single-node subtrees. Subtrees are then combined into larger subtrees, until a single tree is formed. In a top-down scheme, all agents begin with the same parent, which selects some children to be subtree roots, and divides the rest among the subtrees. This continues until only single-node subtrees remain.

When agents belong to different administrative domains, constraints often exist on which agents a given agent may accept as children. Thus, a network provider's agents might serve as parents for customers' agents, but not for competitors' agents.

We focus here on the top-down approach. We show that top-down construction requires fewer resources than bottom-up methods, and how administrative policy constraints may be implemented using a top-down method.

The remainder of this paper is organized as follows. Section 2 discusses previous work and proposes new alternatives. Section 3 presents various issues relating to hierarchy construction. Section 4 gives an analysis of top-down versus bottom-up hierarchy construction. Section 5 describes our top-down construction algorithm. Section 6 gives simulation results, and Section 7 covers conclusions and future work.

2 Previous Work

In this section, we describe related work in both distributed and centralized hierarchy construction.

2.1 DNS and X.500

Hierarchies in typical name services such as DNS [5] and X.500 [6] are constructed by manually configuring each agent with a list of potential parents, ordered by preference. An agent then uses the most preferred and reachable parent. Such manual configuration represents a significant administrative burden. It also allows only a limited form of policy, and provides no guarantee that a hierarchy is constructed unless *all* potential parents are configured.

2.2 Landmark hierarchies

Landmark routing [7–10] constructs a hierarchy out of all nodes in a network. A node's address is determined by its line of descent from the root, and packets are forwarded hop-by-hop towards the visible node closest to the destination address. The hierarchy is constructed bottom-up by having each node broadcast advertisements to all other nodes within a given radius. This radius depends on a node's level in the hierarchy, being larger at higher levels. Based on such advertisements, peers elect parents such that the maximum number of children per parent is bounded. As we will show in Section 4, the use of broadcast (or multicast) advertisements can result in significant overhead.

In [9], Tsuchiya describes a policy scheme for Landmark routing which requires configuring each node on the boundary of an administrative area. This allows broadcast scopes to be constrained by boundaries at various levels, so that a single subtree is built within a given boundary. While this technique may be reasonable for hierarchies where every node is a participant, it is less applicable to constructing hierarchies of distributed agents, since it requires configuration of nodes which are not part of the hierarchy. It is, however, applicable when the same policy constraints apply to multiple protocols, so that similar hierarchies are desirable.

2.3 Area hierarchies

A node's address in an *area hierarchy* [11–13] also corresponds to its position in the hierarchy. In an area hierarchy, however, a node is only aware of its children, its siblings, and the siblings of its direct ancestors; it is not necessarily aware of other nodes which are topologically close. Hagouel [13] and Shacham [12] both discuss general approaches to managing area hierarchies, although neither give detailed protocols or significant analysis. Their schemes both require that a pre-designated "primary" node oversee the formation of each cluster at any level. The primary node must be chosen either administratively or by running some suitable election algorithm.

In Shacham's scheme, non-primary nodes ask to join a cluster by contacting its primary node, which may refuse the request. In Hagouel's scheme, the primary node seeks out potential cluster members. The procedure is then repeated for the next higher layer by selecting new primary nodes. Area hierarchies are thus also constructed bottom-up.

2.4 Clustering algorithms

A number of clustering algorithms have been developed in other domains, such as pattern recognition, where the problem is to identify clusters of points based on their coordinates. Such clustering algorithms may be either agglomerative, working bottom-up from individual points by merging clusters, or divisive, working top-down from the entire space by successively dividing clusters. These algorithms [4] generally require one or more of the following:

- Construct a minimal spanning tree and then partition it in some way [1]. In networks, minimal cost trees spanning some subset of nodes are known as Steiner trees, whose construction is known to be NP-complete [14]. Hence, these techniques are not suitable for use in large dynamic hierarchies of distributed agents.
- Use a region of influence for each node, usually defined by distance (e.g., [2]). In a distributed version, this method would require broadcast advertisements within this region, such as those used by the Landmark scheme as described above.
- Find the k-nearest neighbors for each node (e.g., [3]). This approach does not lend itself well to a distributed algorithm for arbitrary topologies, for two reasons. First, there is no simple method of determining one's k-nearest neighbors without broadcast advertisements, and second, with broadcast advertisements, using regions of influence is more natural than finding the k-nearest neighbors.

3 Issues in Hierarchy Construction

We will use the word *hierarchy* synonymously with the word *tree*, so that a single root exists for the hierarchy. The root has no parent, and every other agent has a single parent and a unique path to the root. Constructing a hierarchy is thus equivalent to determining the parent for each agent.

A hierarchy is different from a spanning tree: it has a single root and contains only *participating* nodes (or "agents"), which may not be adjacent. Hierarchy construction needs no special support in intermediate routers. In this section, we discuss several issues which directly affect the hierarchy construction process.

3.1 Bound on the number of children

An unbounded number of children per agent typically means unbounded processing overhead, both for hierarchy maintenance and use (e.g., nameserver lookups).

Bounding the number of children per agent also involves tradeoffs. A higher bound can increase overhead at the parent, but a lower bound deepens the hierarchy, potentially slowing its use. For example, a nameserver lookup may have to be forwarded through more levels in the tree. Specifically, an m-ary tree of N nodes has between $\log_m N$ and N levels. The number of levels in an N-node tree is inversely proportional to $\log m$ in the best case, and remains constant in the worst case. Thus, increasing m can, at best, only provide a very slow (proportional to $1/\log m$) decrease in levels.

3.2 Root/Neighbor discovery method

In a top-down scheme, all agents must first locate the root. In bottom-up schemes, agents must first discover their neighbors so that merge operations may proceed. In either case, four choices exist for the discovery method:

Static configuration Nodes may be statically configured with a prioritized list of possible root agents in a top-down scheme, or with a list of neighbors for each level in a bottom-up scheme. Although static configuration is administratively burdensome, it is currently used by DNS and X.500.

Scoped broadcast ("Push") For root discovery, the scope is simply the entire network, since all agents need to discover the same root. For neighbor discovery, the scope can be set based on hierarchy level, as in the Landmark [7] scheme. Thus, the higher up in the hierarchy an agent is, the farther away its neighbors may be. For example, if level 0 indicates a leaf and the scope is specified in network hops, the scope might be given by $2^{(level+1)}$. Or, if well-defined borders already exist, the scope might be defined by the area enclosed within level-i borders.

Pull All agents poll some location, which maintains common state. This requires knowledge of a location to poll. Such knowledge must be discovered using one of these four methods, resulting in a recursive problem known as the "Bootstrap Problem" [15]. Thus, while this scheme can be useful for decreasing the amount of information which is pushed or configured, it is not a complete solution without one of the other three options.

Scoped pull A request is broadcast within a given scope, and one or more agents reply. This avoids the bootstrap problem, and trades the overhead of periodic pushes for the overhead of on-demand pushes (pushing a request).

Scheme	Mean CPU cost	Mean link cost
Static	0	0
Bottom-up	O(visibleagents)	$O(\rho * R * N/E)$
Top-down, static root	O(children)	$O(parentdist * N/E)$
Top-down, dynamic rt.	O(children)	$O(parentdist * N/E)$

Table 1: Steady-state Overhead

3.3 Parent election scheme

In a bottom-up scheme, a distributed election algorithm is used by siblings to determine their parent. In a top-down scheme, a parent uses a centralized election mechanism to designate subordinates which are to be parents of subtrees.

The choice of agent to be the parent can be based upon any criteria, including the following ones, which are typical:

- The winner is determined without respect to any properties of the agent. For example, the winner may be picked at random, or be the first one seen by the root in a top-down scheme [20].
- The agent with the highest (or lowest) value of some attribute (e.g., address, or priority) wins. This is referred to as the "Bully" method [16].
- The agent which is the most "centralized" with respect to the others wins, so overhead is minimized. A comparison of such election algorithms can be found in [17].

4 Analysis of Distribution Mechanisms

We now turn to an analysis of the costs of various classes of algorithms. To create and maintain a distributed hierarchy, agents must keep some amount of state, and exchange some amount of control data via network messages. We are therefore interested in the memory requirements and processing costs at individual agents, as well as the network bandwidth requirements. For simplicity, we will assume that in all cases, the steady-state maintenance cost of static configuration is zero. We will estimate the bandwidth requirements of hierarchy construction protocol messages by counting each link traversed by a message as one bandwidth unit.

Consider a bottom-up scheme employing scoped multicast, and let v be the average number of agents visible from any agent. The CPU cost of the algorithm is $O(1)$ to send a message, and $O(v)$ to receive (and possibly forward) messages from each other visible agent. Since multicast messages follow a distribution tree rooted at the sender, each multicast message traverses one link per node within the sender's scope. Let R be the average number of nodes within the scope of an agent. (Note that $R \geq v$, since v also represents the average number of *agents* within the sender's scope.) If there are A agents in a network of N nodes and E edges, then the total bandwidth used is $O(RA)$, and the average link bandwidth is $O(RA/E) = O(\rho RN/E)$, where $\rho = A/N$ represents the agent density.

For top-down schemes with a statically-configured root using maintenance messages unicast to the parent, the only costs are those associated with the unicast messages, and the unicast messages sent by the parent in response. The CPU cost of these is $O(1)$ to send the agent's own message, and $O(|C|)$ to receive and reply to messages from the set C of children. The total bandwidth required is thus proportional to the number of agents times the mean distance each message travels, and the average link cost is given by:

$$O(\text{mean distance to parent} * N/E)$$

Top-down schemes which multicast root advertisements incur additional overhead. The CPU cost added is $O(1)$ to receive and forward (or originate) the root's advertisement, and the link cost is $O(1)$ as well. Table 1 summarizes the costs discussed. From this analysis, we observe that the steady-state overhead of allowing a dynamic root is negligible. Such a scheme could be used whenever broadcast capability exists. Since all children are visible to a parent in any scheme, the top-down approach always has lower CPU overhead than the bottom-up approach.

For the link cost, we observe that with a scoped multicast scheme, where the scope must include the parent, the advertisement will be visible to at least all the nodes between the originator and the parent (*parentdist* nodes). Thus, the top-down scheme usually has less overhead than the bottom-up scheme; the only time the converse could be true is if the average parent distance in the top-down scheme were significantly higher than in the bottom-up scheme.

Taking our two metrics together, a top-down scheme will typically consume less resources than a bottom-up scheme.

5 Top-Down (TDH) Construction

Given a collection of agents, we now present a top-down algorithm for hierarchy construction, which we call TDH. We begin with a general overview of TDH.

All agents use a simple election to elect a single root, which is then known to all agents. Each agent k wishing to participate in the hierarchy applies to the root, which either accepts k as its own child, or redirects it to one of its children which the root knows is willing to accept k as child. Agent k then contacts this child, which again either accepts k as a child, or redirects it to one of its own children. This process continues until some agent accepts the applicant as child. This process always terminates, since an applicant is never redirected to a child unless the parent knows that the child is willing to accept the applicant. We initially assume that the root agent has enough resources available to handle all such first-time queries. In Section 5.5 we will explore optimizations which will relax this assumption.

5.1 Integrating policy constraints

In practice, hierarchies must often be configured to conform to policy constraints. For example, some agents may not be willing or allowed to accept certain other agents as children. Typically, policies tend to group agents into address ranges, and tend to accept or reject such ranges, since address ranges tend to define administrative domains. We handle policy issues by representing acceptable address range sets as bit-string prefixes. We use the term *policy prefix* to denote both an address range as well as the bit string representing that range.

We begin with the following definitions. Let a_i denote the address of an agent i. Let *Parent*(i) denote the agent that agent i believes to be its parent, and let *Children*(i) denote the set of agents that agent i believes to be its children. A *policy prefix* $\mathbf{P}_i$, is the set of addresses of agents that agent i may accept as children. That is, $c \in Children(i) \Rightarrow a_c \in \mathbf{P}_i$. In bit-string form, if i is a parent and c is its child, $\mathbf{P}_i$ is a prefix of the bit string a_c.

Let every agent i be configured with a *maximal* policy prefix $\mathbf{P}_i^{max}$, that represents policy, and have an *active* policy prefix $\mathbf{P}_i \subseteq \mathbf{P}_i^{max}$ determined dynamically in the process of hierarchy construction according to the rules below. If agent c is a child of i, $(\mathbf{P}_i/2)_c$ denotes the half of i's prefix set that includes agent c's address, and is defined as follows. We know that if $\mathbf{P}_i$ is a string of k bits, $p_1p_2\cdots p_k$, then a_c must have the form $p_1p_2\cdots p_kc_{k+1}\cdots c_n$. Now, $(\mathbf{P}_i/2)_c$ is obtained by extending $\mathbf{P}_i$ with c_{k+1}. Clearly, $(\mathbf{P}_i/2)_c$ covers half the addresses covered by $\mathbf{P}_i$, and includes a_c.

A child's maximal policy prefix is a subset of the maximal policy prefix of its parent, and a child's active prefix is a proper subset of its parent's active prefix. In bit-string representation, a child's active prefix string extends the parent's active prefix string.

The algorithm we describe in this section conforms to the following invariants:

I1. A parent may not accept any agent as child whose maximal prefix is not covered by its own maximal prefix. More formally, $c \in Children(p) \Rightarrow \mathbf{P}_c^{max} \subseteq \mathbf{P}_p^{max}$

I2. A child's active prefix is the intersection of its maximal prefix, and the half of its parent's active prefix which covers the child's own address. That is: $c \in Children(p) \Rightarrow \mathbf{P}_c = \mathbf{P}_c^{max} \cap (\mathbf{P}_p/2)_c$

I3. The root's active prefix is identical to its maximal prefix ($\mathbf{P}_{root} = \mathbf{P}_{root}^{max}$).

I4. Any agent which does not know its parent initially assumes that it is itself the root.

Finally by transitivity of $a_{child} \in \mathbf{P}_{parent}$, we have the derived invariant: $a_i \in \mathbf{P}_i \Rightarrow a_i \in \mathbf{P}_{Root(i)}$. Thus, in steady state, the address of an agent is always within the active prefix of the agent it believes to be the root. This also implies that for a single hierarchy to exist, some agent's maximal prefix must cover the addresses of all other agents which are reachable. That is, $\exists r : \forall i(a_i \in \mathbf{P}_r^{max})$. We call such an agent a "legal root".

We now describe a top-down hierarchy construction algorithm with dynamic root discovery. This algorithm consists of two parts: determining the root, and determining one's parent. We will describe these two parts in turn. Figure 1 illustrates the state transition diagram used by each agent in the discussions below. Pseudocode is available in [18].

5.2 Determining the Root

If multicast or broadcast mechanisms are not available, then each agent must be configured with the address of one or more other agents to use initially as root. This strategy is not an undue administrative burden, since all agents can be configured identically, and such information could be

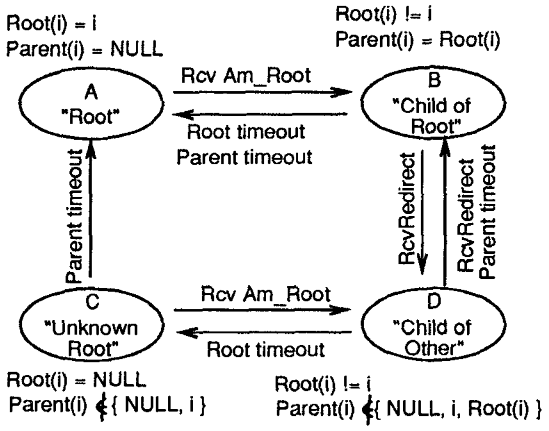


Figure 1: State Transition Diagram

distributed with the application, or looked up in a global directory (as when resolving a well-known hostname).

When multicast is available, root election and discovery can be accomplished through an election procedure similar to the "Bootstrap Router" election used by PIM-SMv2 [15]. The root is chosen by a simple Bully election: the agent with the shortest maximal policy prefix wins, with ties broken by highest preconfigured "priority" value, and then by lowest address. This operates as follows.

An agent starts up in state A (see Figure 1) as the root of its own subtree, and starts a periodic Advertisement-Timer with a random delay. Agents in state A broadcast `Am_Root` messages to all other agents at each timer interval. Receiving this message causes all agents except the most preferred agent (according to the Bully election rules) to transition to state B, and start their Root-Timer.

An agent in any other state uses its Advertisement-Timer to send periodic `Am_Child` messages directly to its parent, thus refreshing the parent's entry for the child. Any child entries not refreshed are eventually deleted by the parent.

The Root-Timer is restarted whenever an `Am_Root` message is received from the Root agent. If the Root becomes unreachable, this timer will expire at each agent, and agents in states B and D will move to states A and C respectively. Agents moving to state A do so from state B, where they were the root's immediate children. These assume the role of root, and begin to broadcast `Am_Root` messages. The advertisements from the most preferred agent in state A again override all other `Am_Root` messages. A new root will thus be elected, allowing the hierarchy to adapt to the failure of the root agent.

5.3 Determining One's Parent

Once the root is known and an agent n enters state B, it may proceed as follows to determine its parent, and thus its place in the hierarchy. The algorithm is iterative, and the agent n successively refines its notion of who its parent is.

The agent n initially takes the root to be its parent, and starts a timer (Parent-Timer). At each iteration, n sends an **`Am_Child`** message containing its own address and maximal prefix to its current parent. The potential parent must then decide whether to accept agent n as a child, or whether n properly belongs in a subtree under one of its current chil-

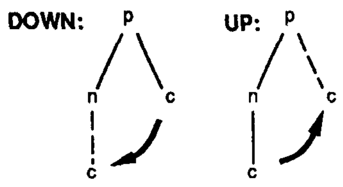


Figure 2: Redirection

dren. This determination is made by comparing n's address and maximal prefix with its own, and with those of its children. An agent p is a legal parent of an agent n if p's maximal prefix covers n's maximal prefix, and p's active prefix covers n's address. If any child c of p is a legal parent for n, the agent p redirects n to its child c by sending a `Redirect` message to n. Otherwise, the parent p will accept the agent n as its own child by responding with an `Am_Parent` message. As we will show below in Section 5.4, at most one child c will be a legal parent of the new agent n.

If the new agent n is accepted as a child, its parent p must also check to see whether n is a legal parent for any of p's current children. If so, the parent sends `Redirect` messages to such children, redirecting them down to n (Figure 2).

A child receiving a `Redirect` sets its parent to be the agent indicated in the message, restarts its Parent-Timer, and repeats the procedure above for finding its parent.

A child receiving an `Am_Parent` message in response to its periodic `Am_Child` message will restart its Parent-Timer. If not explicitly restarted, the Parent-Timer will eventually expire, causing an agent to move from state B, C, or D, to state A, A, or B, respectively, as shown in Figure 1. Agents moving to state A in this way repeat the root election procedure described in Section 5.2 by broadcasting `Am_Root` messages unless suppressed by a more preferred agent's `Am_Root` message. Agents moving to state B repeat the parent discovery procedure described above.

A child receiving an `Am_Parent` message will also set its active prefix according to invariant **I2** of Section 5.1. If this causes an agent with children to lengthen its active prefix, invariant **I1** may be invalidated. When this occurs, the agent sends a `Redirect` message to all children for which **I1** is invalid, redirecting them up to its own parent (Figure 2), and removing them from its list of children.

5.4 Discussion

The parent selection scheme in Section 5.3 generates a hierarchy which is sensitive to the order of addition. This difficulty could be addressed by having a parent use some criteria (Section 3.3) for performing redirection, when a new agent and an existing child can both be parents for each other. This approach causes additional reorganization, however, which is often undesirable.

In the rest of this section, we will show that children's active policy prefixes do not overlap, and that the algorithm above generates a stable hierarchy in steady-state. We will do this in the form of three theorems, below.

Theorem 1 (Children don't overlap): *If agents i and j are sibling agents, then their active policy prefixes $\mathbf{P}_i$ and $\mathbf{P}_j$ are disjoint.*

Proof: Invariant **I1** of Section 5 tells us that a parent p initially accepts a child c if $\mathbf{P}_c^{max} \subseteq \mathbf{P}_p^{max}$. Also, $a_c \in \mathbf{P}_p$. Since i is not a child of j, either $\mathbf{P}_i^{max} \not\subseteq \mathbf{P}_j^{max}$ or $a_i \notin \mathbf{P}_j$. Similarly, since j is not a child of i, either $\mathbf{P}_j^{max} \not\subseteq \mathbf{P}_i^{max}$ or $a_j \notin \mathbf{P}_i$. (Otherwise, either i or j would have been redirected to the other by their common parent.)

We must therefore consider four cases. We begin by noting the following property of prefixes: if A and B are prefixes, then $A \cap B \neq \emptyset \Rightarrow A \subset B$ or $B \subseteq A$.

In the first case, $\mathbf{P}_i^{max} \not\subseteq \mathbf{P}_j^{max}$, and $\mathbf{P}_j^{max} \not\subseteq \mathbf{P}_i^{max}$. From the property of prefix sets noted above, we can conclude that $\mathbf{P}_i^{max} \cap \mathbf{P}_j^{max} = \emptyset$. Finally, since $\mathbf{P}_i \subseteq \mathbf{P}_i^{max}$ and $\mathbf{P}_j \subseteq \mathbf{P}_j^{max}$, the theorem immediately follows.

We show the remaining cases by contradiction.

In the second case, $a_i \notin \mathbf{P}_j$ and $a_j \notin \mathbf{P}_i$. If $\mathbf{P}_i \cap \mathbf{P}_j \neq \emptyset$, then from the prefix set property, either $a_i \in \mathbf{P}_i \subset \mathbf{P}_j$ or $a_j \in \mathbf{P}_j \subseteq \mathbf{P}_i$, and we have a contradiction.

In the third case (and the fourth, by symmetry), we have $\mathbf{P}_i^{max} \not\subseteq \mathbf{P}_j^{max}$ and $a_j \notin \mathbf{P}_i$. If $\mathbf{P}_i \cap \mathbf{P}_j \neq \emptyset$, then either $\mathbf{P}_i \subset \mathbf{P}_j$ or $\mathbf{P}_j \subseteq \mathbf{P}_i$. If $\mathbf{P}_j \subseteq \mathbf{P}_i$, then $a_j \in \mathbf{P}_i$, a contradiction. On the other hand, if $\mathbf{P}_i \subset \mathbf{P}_j$, then $\exists x \in \mathbf{P}_j$ such that $x \notin \mathbf{P}_i$. We next observe that $(\mathbf{P}_p/2)_i = (\mathbf{P}_p/2)_j$ (if not, $\mathbf{P}_i \cap \mathbf{P}_j = \emptyset$). We also know that $\mathbf{P}_j^{max} \subset \mathbf{P}_i^{max}$ since $\mathbf{P}_i \cap \mathbf{P}_j \neq \emptyset$ and $\mathbf{P}_j \not\subseteq \mathbf{P}_i$. Thus, we have $x \in \mathbf{P}_j = \mathbf{P}_j^{max} \cap (\mathbf{P}_p/2)_j \subseteq (\mathbf{P}_p/2)_j$, and $x \in \mathbf{P}_j^{max} \subset \mathbf{P}_i^{max}$, so $x \in \mathbf{P}_i^{max} \cap (\mathbf{P}_p/2)_i = \mathbf{P}_i$, a contradiction. □

The above theorem has certain implications on the number of children per parent. Specifically, if children have maximal policy prefixes which are at least as inclusive as the parent's active policy prefix, then the size of the children's active prefix set will be half the parent's active prefix set. Theorem 1 says that the number of children per parent will then be at most two in this case. A lower number of children has the advantage of keeping the processing requirements per parent low, at the expense of more levels in the tree.

We define a *stable* agent as one whose parent will not change in the absence of failures, message loss, or configuration changes. We define a stable *hierarchy* as one in which all agents are stable. The following theorems show that our method generates stable hierarchies.

Theorem 2 (Stability): *In steady-state, a stable hierarchy exists if a legal root is in state A, all other agents are in states B and D, and $Parent(i) \neq NULL \Rightarrow i \in Children(Parent(i))$.*

Proof: We first note that since active prefixes are strictly nested (Invariant **I2**), loops cannot exist in steady state. Also, an agent may only change its parent in response to a Parent-Timer or Root-Timer expiration, or in response to a `Redirect` or `Am_Root` message.

If a legal root is in state A, it has no parent. Thus, it will never leave state A (i.e., never set its parent) since it will never receive an `Am_Root` from agents in other states. In the absence of message loss, other agents will then never experience a Root-Timer expiration, since they will receive `Am_Roots` from this legal root. Similarly, `Am_Parent` responses to periodic `Am_Child` messages will maintain parent state in children, so that Parent-Timer expirations do not occur. Finally, **Redirects** are only sent when a parent hears from a new child (i.e., when $i \notin Children(Parent(i))$), or when a parent's active prefix changes. The first case is covered by the condition ($Parent(i) \neq NULL \Rightarrow i \in Children(Parent(i))$) from the theorem statement. For the latter, we observe that the definition of the active prefix (**I2**) says that in steady state, a child's prefix will not change unless its parent's prefix changes. Since there are no loops, and the root's active prefix never changes (**I3**), then by induction, every agent's active prefix remains constant. □

Theorem 3 (Convergence): *The TDH algorithm converges to a stable hierarchy if a legal root exists.*

Due to space limitations, we have omitted the details of the proof, which can be found in [18]. Briefly, it can be shown by contradiction that in steady state, the most preferred agent will be in state A, all others will be in states B and D, and $Parent(i) \neq NULL \Rightarrow i \in Children(Parent(i))$. This is done by showing that every other case is not stable, but converges to the above conditions. Thus, by Theorem 2, a stable hierarchy will emerge. If a legal root does not exist, TDH will instead result in multiple independent hierarchies, each rooted at a legal root for all agents in its hierarchy.

5.5 Optimizations

If a large number of agents start up simultaneously, the root could be overwhelmed with messages. In this section, we investigate optimizations which relax the assumption that the root agent has enough resources to respond to all agents' initial requests for a parent.

We know from Section 5.3 that an agent sends a message to the root in three cases: upon starting up, when its parent state expires (i.e., its parent dies or becomes unreachable), and periodically, if the root has accepted it as a child. The last case poses no problems, since the root, being a parent, must see the same overhead as any other parent.

The message load at the root from the other two situations may be reduced by allowing agents to contact agents other than the root. If an orphan sets its parent to some agent other than the root, it moves to state D, rather than B.

The algorithm runs correctly even if a non-legal parent is initially chosen as parent by some agent, since the chosen parent will respond with a **Redirect** message, redirecting the potential child up the hierarchy (Figure 2). This process will continue until a legal parent is found, after which any downward redirections occur as described in Section 5.3.

It only remains to efficiently inform all agents of *any* other agents already in the hierarchy to contact, so that the responsibility of replying to orphan agents is distributed. Possible methods (making different tradeoffs between the amount of global resources required and the degree to which the load on the root is reduced) correspond to the root/neighbor discovery schemes discussed in Section 3.2, namely:

Static Configuration: This can be administratively burdensome to maintain. It also does not ensure that any of the addresses will be in the hierarchy.

Scoped Broadcast ("Push"): When multicast or broadcast facilities are available, some agents which already have parents may also broadcast announcements. Orphans can then select any legal parent as the agent to query. The choice of whether an agent should send announcements may be made based on any of several criteria. For example, it could be based on static configuration, or active prefix length (limiting broadcasters to those at the highest levels). Alternatively, an RTP-like scheme [19] could be used in which all agents broadcast, the broadcast periods being scaled back with the number of agents so that the overall frequency (and hence the bandwidth usage) remains the same.

Pull: As described in Section 3, this option is not complete, requiring one of the other three schemes in addition.

Scoped Pull: When multicast or broadcast facilities are available, an agent can initially employ an "expanding ring multicast" scheme to locate a nearby agent already in the hierarchy. This is done by successively broadcasting queries with increasing time-to-live (i.e., hop-count) until a potential parent responds.

6 Simulation

In this section, we will present the results of simulating the performance of various hierarchy construction algorithms, according to the criteria described below.

We simulated two schemes. The first scheme was the top-down scheme described in Section 5, and is labelled TD. The second was a simple Landmark-like bottom-up scheme employing scoped multicast with a radius of two for leaves, doubling at each level up. To select between several equal parents, we could use any deterministic algorithm which operates as a purely local decision with negligible overhead. Our simulations chose the closest parent, using HRW [20] to break ties. HRW allows children to deterministically choose parents so that whenever several children may choose from the same set of parents, each parent gets roughly the same number of such children. The number of children per parent was unbounded. In both schemes, the periodic message interval was 60 seconds and all timer expirations were set to 130 seconds.

We use the random graph model of Waxman [21], which randomly places nodes over a Cartesian coordinate system, and creates edges as described in [21].

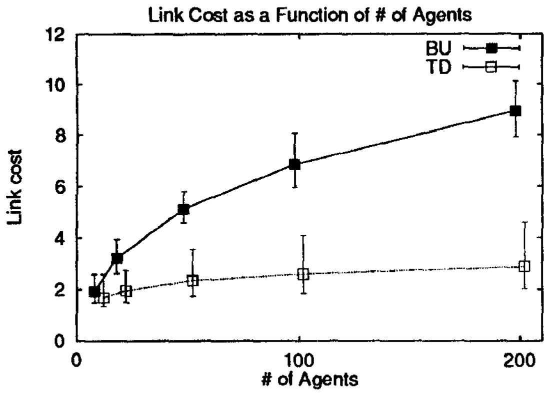
6.1 Parameters of interest

We are first interested in the steady-state distribution of CPU overhead (as summarized in Table 1). We are also interested in the distribution of the number of children, since having too many children can adversely affect an agent's ability to participate in the application *using* the hierarchy.

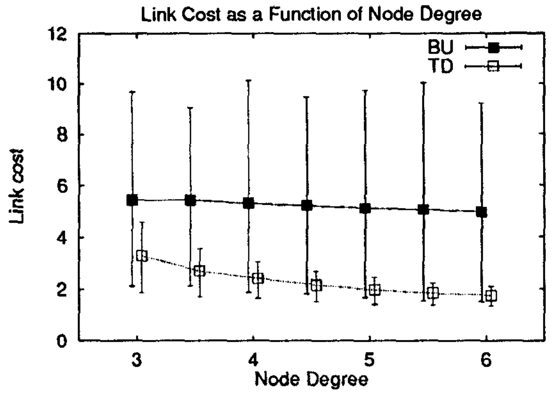
We are also interested in the total amount of bandwidth consumed in steady-state. We measure this in terms of "link cost", or the average number of messages per link per periodic update interval. Finally, we are interested in the convergence time for adapting to failures of intermediate agents in the hierarchy.

6.2 Basic simulations

In our first set of simulations, all nodes were agents, no failures occurred, and no policy limitations existed (i.e., ev-

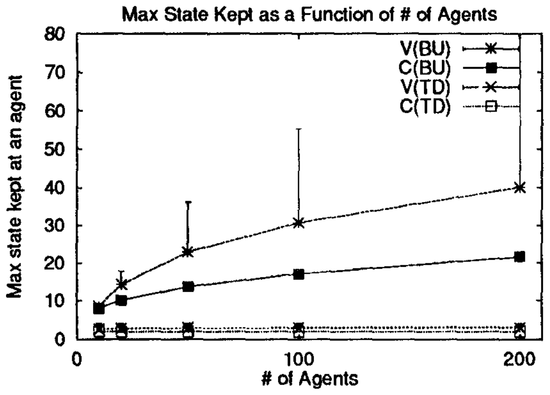


(a) Effects of Number of Agents



(b) Effects of Node Degree

Figure 3: Bandwidth Usage



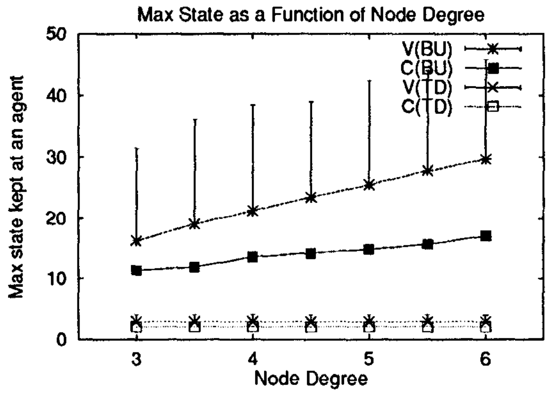


Figure 4: Agent Resource Requirements

ery agent was a legal parent of every other agent). Node addresses were random 32-bit addresses in a flat domain. We ran ten simulations of each algorithm of interest, for each combination of network size and node degree. These combinations were derived from seven different node degrees between 3 and 6, and five different network sizes, of between 10 and 200 nodes.

Since this yields a relatively large number of combinations, plotting all our results tended to create very cluttered graphs. We observed, however, that each algorithm tended to give rise to a cluster of curves, each curve corresponding to a choice of the above parameters. We therefore used errorbars to capture the range of y-axis values across the cluster of curves for a given x-axis value. The average of these values was also plotted as a curve within this range.

Figure 3 shows how the number of agents and average node degree affect the bandwidth requirements in steady-state. Both algorithms were run on the same set of graphs. In Figure 3(a), each point represents an average over 70 trials with varying node degrees. As usual, errorbars indicate the maximum and minimum values observed over this range. For ease of comparison, the results of the two algorithms are slightly offset along the X-axis. We see that the link cost of the bottom-up scheme increases significantly as the number of agents increases, since the number of visible agents increases as more levels are added. The link cost of the top-down scheme remains relatively low, increasing only slightly with the average parent distance.

Since the average node degree $= 2E/N$, we would naturally expect from Table 1 that the link costs would decrease somewhat as the node degree increases. This is confirmed by Figure 3(b), where each point represents an average over 50 trials with varying network sizes. For the bottom-up scheme, however, the expected decrease is mostly counterbalanced by an increase in the number of nodes within an agent's scope. In general, the more sparsely-connected the network is, the higher the bandwidth required to construct a hierarchy in either scheme.

Figure 4 shows the steady-state distribution of agent resource requirements. For each algorithm, the following quantities are shown:

- maximum number of children at any agent, averaged over all trials (note that the average number of children per agent is always $(N-1)/N$). These are the "C" lines.
- average number of visible agents (the "V" lines).
- maximum number of agents visible at any agent (averaged over all trials). This quantity is indicated by the tops of the errorbars connected to the "V" lines.

Clearly, the top-down scheme requires the least state (and hence the least CPU processing overhead in steady state). The number of children is bounded at two, since in Invariant

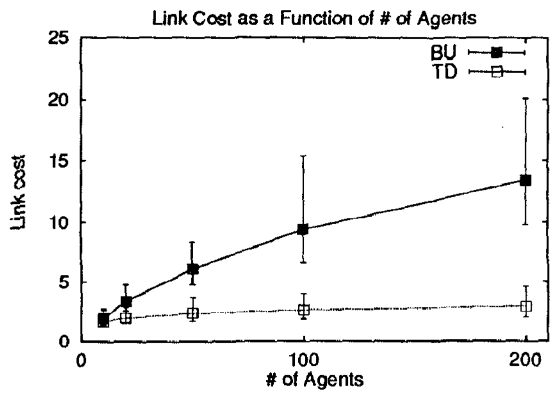


Figure 5: Bandwidth Usage with Policy Constraints

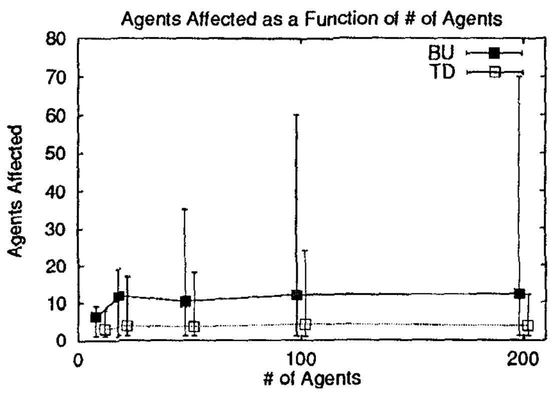


Figure 7: Agents Affected by a Failure

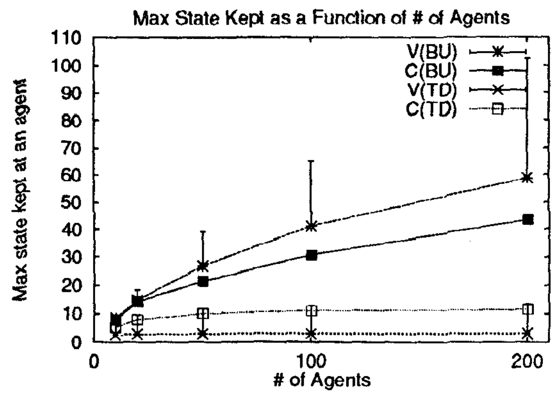


Figure 6: Agent Resource Requirements with Policy Constraints

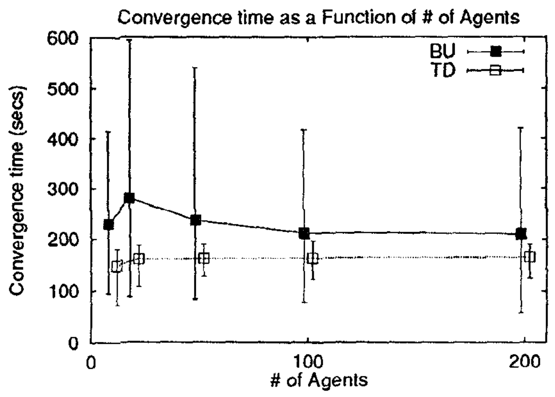


Figure 8: Reconstruction Time

I2 the parent's active prefix controls the children's prefixes. The bottom-up scheme requires more resources as both the number of agents and the average node degree increase.

If we had bounded the number of children in the bottom-up scheme, this would have been reflected by a lower "C" line. However, this would have increased the number of levels, and hence the radii of many agents. As a result, the link cost, build time, and number of visible agents would have risen accordingly. Our simulation thus shows the worst case for "C", but the *best* case for all other metrics in Figures 3–4. Bounding the number of children in a bottom-up scheme would thus increase the advantage of using TDH instead.

6.3 Adding Policy

In the next set of simulations, we added administrative policy constraints by assigning each agent a maximal policy prefix. We randomly assigned each agent a preference value between 0 and 24, and then associated preference values with mask lengths between 0 and 24 bits long, giving the lowest preference value a 0-bit mask.

The top-down scheme then used the resulting maximal prefix as described in Section 5. The bottom-up scheme used this prefix by not accepting as children any visible agents whose addresses fell outside an agent's maximal prefix.

Figure 5 shows the results in terms of link cost. We see that the cost of bottom-up schemes has increased by up to 100% compared to Figure 3, while the cost of the top-down scheme is relatively unaffected.

Figure 6 shows the steady-state distribution of agent resource requirements. Comparing this to Figure 4, we see that the CPU and memory requirements have also increased by up to 100% as a result of adding policy constraints. We also note that the maximum number of children in the top-down scheme has increased. This is because the active prefixes are often controlled by the agents' maximal prefixes rather than their parents' active prefixes. This allows more than two children to exist without overlapping prefixes.

6.4 Adapting to Failures

The results in this section were derived by introducing failures into the simulations in Section 6.3. Starting from a stable hierarchy with policy constraints, a non-leaf agent was selected at random to fail, such that it was still possible to find a single connected hierarchy. We then measured the elapsed time and number of messages required to converge to a new stable hierarchy.

Figure 7 shows the distribution of the number of agents whose parent changed as a result of the failure of one agent. In the bottom-up scheme, the failure sometimes caused ripple effects across much of the network. The top-down scheme on the other hand, was much better at localizing changes.

Figure 8 shows the convergence time, starting from the time at which the intermediate agent died. We observe that the convergence time of both schemes is relatively unaffected as the number of agents vary, and that again the top-down scheme is better.

Figure 9 shows the distribution of the number of messages received by each node during this convergence. The curves

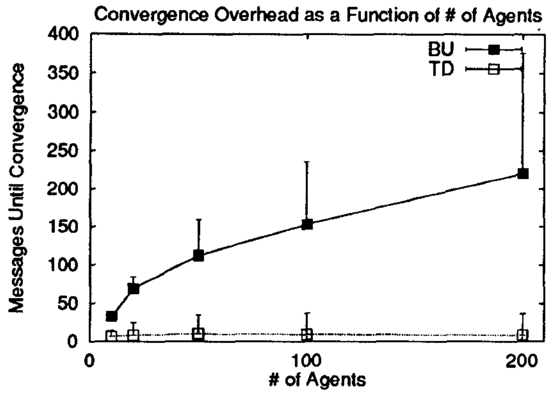


Figure 9: Convergence Overhead

indicate the average number, and the tops of the errorbars indicate the maximum number of messages received by any node (averaged over all trials). Intuitively, we would expect the number of messages to be related to the number of children affected by the failure of a parent. Comparing Figures 9 and 6 shows the results to indeed be similar. Secondly, comparing figures 8 and 9 gives an indication of the average and maximum CPU overhead imposed by a node failure. From this comparison, we see that the rate at which messages are received is much lower in the top-down scheme.

7 Conclusions

Many applications, such as nameservice [5, 6] and network management [18], use hierarchies of distributed agents. In this paper, we presented a solution to the important problem of constructing hierarchies in distributed fashion. We presented a top-down approach to this problem, and described a specific top-down algorithm (TDH) with a number of desirable properties. We compared the top-down and bottom-up approaches through both analysis and simulation, and showed that the top-down approach requires less resources than the bottom-up approach. Our top-down approach also results in less overhead caused by the addition of administrative policy constraints.

The TDH algorithm keeps the number of children per parent low, which in turn keeps the overhead at each agent low. The state and CPU overhead of hierarchy maintenance in the top-down scheme increases linearly with the number of children, while the overhead of additional levels imposed on the application using the hierarchy decreases much more slowly. Thus, fewer children are better.

One possible disadvantage of the top-down approach is that it can impose a large resource requirement on the root when a large number of agents either start up or lose their parent simultaneously. We described several possible ways to counter this effect.

In conclusion, we believe that our TDH algorithm is very usable in applications that must run over large distributed environments like the Internet.

References

[1] C. Zahn. Graph-theoretical methods for detecting and describing gestalt clusters. *IEEE Trans. Computers*, 20:68–86, 1971.

[2] W. Koontz, P. Narendra, and K. Fukunaga. A graph-theoretic approach to nonparametric cluster analysis. *IEEE Trans. Computers*, 25(9):936–944, Sep. 1976.

[3] Riichiro Mizoguchi and Masamichi Shimura. A nonparametric algorithm for detecting clusters using hierarchical structure. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 2(4):292–300, July 1980.

[4] R. Urquhart. Graph theoretical clustering based on limited neighbourhood sets. *Pattern Recognition*, 15(3), 1982.

[5] Paul Mockapetris. Domain names - concepts and facilities, Nov. 1987. RFC-1034.

[6] Gerald W. Neufeld. Descriptive names in X.500. In *Proc. ACM SIGCOMM'89*, pages 64–70, 1989.

[7] Paul F. Tsuchiya. The landmark hierarchy: A new hierarchy for routing in very large networks. In *Proceedings of the ACM SIGCOMM*, 1988.

[8] Paul F. Tsuchiya. The landmark hierarchy: Description and analysis. Technical Report MTR-87W00152, MITRE Corporation, June 1987.

[9] Paul F. Tsuchiya. Landmark routing: Architecture, algorithms, and issues. Technical Report MTR-87W00174, MITRE Corporation, Sep. 1987.

[10] Paul F. Tsuchiya and Ron Zahavi. Landmark routing algorithms: Analysis and simulation results. Technical Report MTR-89W00277, MITRE Corporation, Dec. 1989.

[11] L. Kleinrock and F. Kamoun. Hierarchical routing for large networks: Performance evaluation and optimization. *Computer Networks*, 1:155–174, 1977.

[12] Nachum Shacham and Jil Westcott. Future directions in packet radio architectures and protocols. *Proceedings of the IEEE*, 75(1):83–99, Jan. 1987.

[13] Jacob Hagouel. Issues in routing for large and dynamic networks, May 1983. Ph.D. Thesis, Columbia University.

[14] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, June 1988.

[15] Estrin, Handley, Helmy, Huang, and Thaler. A dynamic bootstrap mechanism for rendezvous-based multicast routing, 1997. Submitted to IEEE/ACM Trans. Networking.

[16] H. Garcia-Molina. Elections in a distributed computing system. *IEEE Trans. Computers*, C-31(1):48–59, Jan. 1982.

[17] David G. Thaler and Chinya V. Ravishankar. Distributed center-location algorithms. *IEEE J. Select. Areas in Commun.*, 15(3):291–303, Apr. 1997.

[18] David G. Thaler. An architecture for inter-domain network troubleshooting, 1998. Ph.D. Thesis, University of Michigan.

[19] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A transport protocol for real-time applications, Jan. 1996. RFC-1889.

[20] David G. Thaler and C.V. Ravishankar. Using name-based mappings to increase hit rates. *IEEE/ACM Trans. Networking*, 6(1), Feb. 1998.

[21] Bernard M. Waxman. Routing of multipoint connections. *IEEE J. Select. Areas in Commun.*, 6(9), Dec. 1988.