Hierarchical Inter-Domain Routing Protocol with On-Demand ToS and Policy Resolution*

Cengiz Alaettinoğlu, A. Udaya Shankar

Institute for Advanced Computer Studies
Department of Computer Science
University of Maryland
College Park, Maryland 20742

CS-TR-3299

June 20, 1994

Abstract

Traditional inter-domain routing protocols based on superdomains maintain either "strong" or "weak" ToS and policy constraints for each visible superdomain. With strong constraints, a valid path may not be found even though one exists. With weak constraints, an invalid domain-level path may be treated as a valid path.

We present an inter-domain routing protocol based on superdomains, which always finds a valid path if one exists. Both strong and weak constraints are maintained for each visible superdomain. If the strong constraints of the superdomains on a path are satisfied, then the path is valid. If only the weak constraints are satisfied for some superdomains on the path, the source uses a query protocol to obtain a more detailed "internal" view of these superdomains, and searches again for a valid path. Our protocol handles topology changes, including node/link failures that partition superdomains. Evaluation results indicate our protocol scales well to large internetworks.

Categories and Subject Descriptors: C.2.1 [Computer-Communication Networks]: Network Architecture and Design—*packet networks; store and forward networks*; C.2.2 [Computer-Communication Networks]: Network Protocols—*protocol architecture*; C.2.m [Routing Protocols]; F.2.m [Computer Network Routing Protocols].

* This work is supported in part by ARPA and Philips Labs under contract DASG60-92-0055 to Department of Computer Science, University of Maryland, and by National Science Foundation Grant No. NCR 89-04590. The views, opinions, and/or findings contained in this report are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the Advanced Research Projects Agency, PL, NSF, or the U.S. Government. Computer facilities were provided in part by NSF grant CCR-8811954.

Contents

1 Introduction

A computer internetwork, such as the Internet, is an interconnection of backbone networks, regional networks, metropolitan area networks, and stub networks (campus networks, office networks and other small networks)[1]. Stub networks are the producers and consumers of the internetwork traffic, while backbones, regionals and MANs are transit networks. Most of the networks in an internetwork are stub networks. Each network consists of nodes (hosts, routers) and links. A node that has a link to a node in another network is called a *gateway*. Two networks are *neighbors* when there is one or more links between gateways in the two networks (see Figure 1).

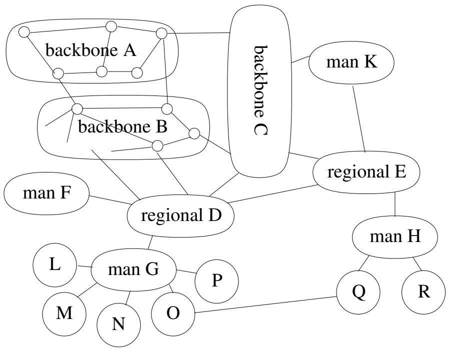

Figure 1: A portion of an internetwork. (Circles represent stub networks.)

An internetwork is organized into *domains*[2]. A domain is a set of networks (possibly consisting of only one network) administered by the same agency. Domains are typically subject to *policy constraints*, which are administrative restrictions on inter-domain traffic [7, 11, 8, 5]. The policy constraints of a domain U are of two types: *transit policies*, which specify how other domains can use the resources of U (e.g. \$0.01 per packet, no traffic from domain V); and *source policies*, which specify constraints on traffic originating from U (e.g. domains to avoid/prefer, acceptable connection cost). Transit policies of a domain are public (i.e. available to other domains), whereas source policies are usually private.

Within each domain, an *intra-domain routing protocol* is executed that provides routes between source and destination nodes in the domain. This protocol can be any of the typical ones, i.e., next-hop or source routes computed using distance-vector or link-state algorithms. To satisfy

[1] For example, NSFNET, MILNET are backbones, and Suranet, CerfNet are regionals.
[2] Also referred to as *routing domains* or *administrative domains*.

type-of-service (ToS) constraints of applications (e.g. low delay, high throughput, high reliability, minimum monetary cost), each node maintains a *cost* for each outgoing link and ToS. The intra-domain routing protocol should choose optimal paths based on these costs.

Across all domains, an *inter-domain routing protocol* is executed that provides routes between source and destination nodes in different domains, using the services of the intra-domain routing protocols within domains. This protocol should have the following properties:

(1) It should satisfy the policy constraints of domains. To do this, it must keep track of the policy constraints of domains [5].

(2) An inter-domain routing protocol should also satisfy ToS constraints of applications. To do this, it must keep track of the ToS services offered by domains [5].

(3) An inter-domain routing protocol should scale up to very large internetworks, i.e. with a very large number of domains. Practically this means that processing, memory and communication requirements should be *much less than linear* in the number of domains. It should also handle non-hierarchical domain interconnections at any level [8] (e.g. we do not want to hand-configure special routes as "back-doors").

(4) An inter-domain routing protocol should automatically adapt to link cost changes and node/link failures and repairs, including failures that partition domains [13].

A Straight-Forward Approach

A straight-forward approach to inter-domain routing is domain-level source routing with link-state approach [7, 5]. In this approach, each router[3] maintains a *domain-level view* of the internetwork, i.e., a graph with a vertex for every domain and an edge between every two neighbor domains. Policy and ToS information is attached to the vertices and the edges of the view.

When a source node needs to reach a destination node, it (or a router[4] in the source's domain) first examines this view and determines a *domain-level source route* satisfying ToS and policy constraints, i.e., a sequence of domain ids starting from the source's domain and ending with the destination's domain. Then packets are routed to the destination using this domain-level source route and the intra-domain routing protocols of the domains crossed.

For example, consider the internetwork of Figure 2 (each circle is a domain, and each thin line

[3] Not all nodes maintain routing tables. A router is a node that maintains a routing table.

[4] referred to as the policy server in [7]

is a domain-level interconnection). Suppose a node in $d1$ desires a connection to a node in $d7$. Suppose the policy constraints of $d3$ and $d19$ do not allow transit traffic originating from $d1$. Every node maintains this information in its view. Thus the source node can choose a valid path from source domain $d1$ to destination domain $d7$ avoiding $d3$ and $d19$ (e.g. thick line in the figure).

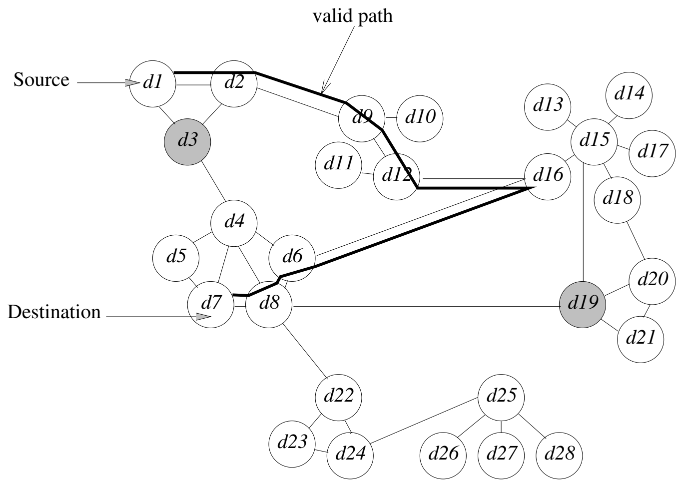

Figure 2: An example interdomain topology.

The disadvantage of this straightforward scheme is that it does not scale up for large internetworks. The storage at each router is proportional to $N_D \times E_D$, where N_D is the number of domains and E_D is the average number of neighbor domains to a domain. The communication cost for updating views is proportional to $N_R \times E_R$, where N_R is the number of routers in the internetwork and E_R is the average router neighbors of a router (topology changes are flooded to all routers in the internetwork).

The Superdomain Approach

To achieve scaling, several approaches based on hierarchically aggregating domains into *superdomains* have been proposed [16, 14, 6]. Here, each domain is a level 1 superdomain, "close" level 1 superdomains are grouped into level 2 superdomains, "close" level 2 superdomains are grouped into level 3 superdomains, and so on (see Figure 3). Each router x maintains a view that contains the level 1 superdomains in x's level 2 superdomain, the level 2 superdomains in x's level 3 superdomain (excluding the x's level 2 superdomain), and so on. Thus a router maintains a smaller view than it would in the absence of hierarchy. For the superdomain hierarchy of Figure 3, the views of two

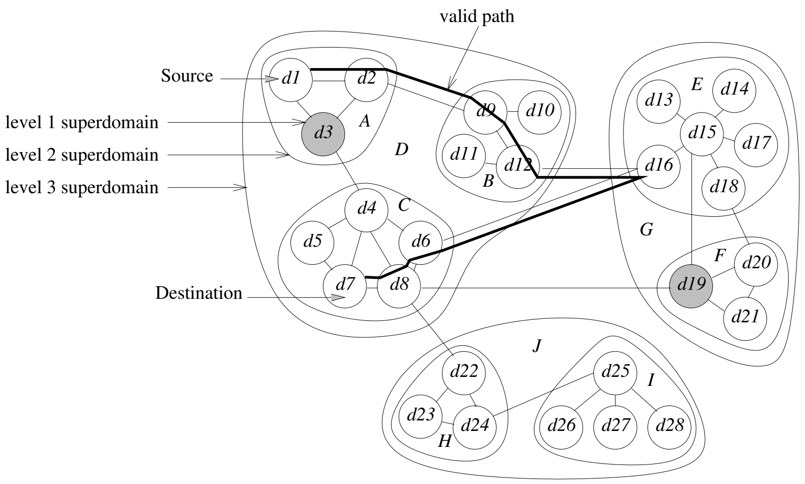

Figure 3: An example of superdomain hierarchy.

routers (one in domain $d1$ and one in domain $d16$) are shown in Figures 4 and 5.

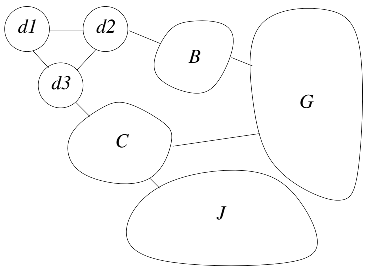

Figure 4: View of a router in $d1$.

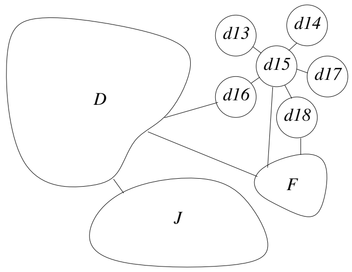

Figure 5: View of a router in $d16$.

The superdomain approach has several problems. One problem is that the aggregation results in loss of domain-level ToS and policy information. A superdomain is usually characterized by a single set of ToS and policy constraints derived from the ToS and policy constraints of the domains in it. Routers outside the superdomain assume that this set of constraints applies uniformly to each of its children (and by recursion to each domain in the superdomain). If there are domains with different (possibly contradictory) constraints in a superdomain, then there is no good way of deriving the ToS and policy constraints of the superdomain.

The usual technique [16] of obtaining ToS and policy constraints of a superdomain is to obtain either a *strong* set of constraints or a *weak* set of constraints[5] from the ToS and policy constraints of

[5] "strong" and "weak" are referred to respectively as "union" and "intersection" in [16]

the children superdomains in it. If strong (weak) constraints are used for policies, the superdomain enforces a policy constraint if that policy constraint is enforced by some (all) of its children. If strong (weak) constraints are used for ToS constraints, the superdomain is assumed to support a ToS if that ToS is supported by all (some) of its children. The intention is that if strong (weak) constraints of a superdomain are (are not) satisfied then any (no) path through that superdomain is valid.

Each approach has problems. Strong constraints can eliminate valid paths, and weak constraints can allow invalid paths. For example in Figure 3, $d16$ allows transit traffic from $d1$ while $d19$ does not; with strong constraints G would not allow transit traffic from $d1$, and with weak constraints G would allow transit traffic from $d1$ to be routed via $d19$.

Other problems of the superdomain approach are that the varying visibilities of routers complicates superdomain-level source routing and handling of node/link failures (especially those that partition superdomains). The usual technique for solving these problems is to augment superdomain-level views with gateways [16] (see Section 3).

Our Contribution

In this paper, we present an inter-domain routing protocol based on superdomains, which finds a valid path if and only if one exists. Both strong and weak constraints are maintained for each visible superdomain. If the strong constraints of the superdomains on a path are satisfied, then the path is valid. If only the weak constraints are satisfied for some superdomains on the path, the source uses a query protocol to obtain a more detailed "internal" view of these superdomains, and searches again for a valid path.

We use superdomain-level views with gateways and a link-state view update protocol to handle topology changes including failures that partition superdomains. The storage cost is $O(\log N_D \times \log N_D)$ without the query protocol. We demonstrate the scaling properties of the query protocol by giving evaluation results based on simulations. Our evaluation results indicate that the query protocol can be performed using 15% extra space.

Our protocol consists of two subprotocols: a **view-query protocol** for obtaining views of greater resolution when needed; and a **view-update protocol** for disseminating topology changes to the views.

Several approaches to scalable inter-domain routing have been proposed, based on the superdomain hierarchy [1, 14, 16, 9, 6], and the landmark hierarchy [18, 17]. Some of these approaches suffer from loss of ToS and policy information (and hence may not find a valid path which exists). Others are still in a preliminary stage. (Details in Section 8.)

One important difference between these approaches and ours is that ours uses a query mechanism to obtain ToS and policy details whenever needed. In our opinion, such a mechanism is needed to obtain a scalable solution. Query protocols are also being developed to enhance the protocols in [9, 6]. Reference [2] presents protocols based on a new kind of hierarchy, referred to as the viewserver hierarchy (more details in Section 8).

A preliminary version of the view-query protocol was proposed in reference [1]. That version differs greatly from the one in this paper. Here, we augment superdomain-level views with gateways. In [1], we augmented superdomain-level views with superdomain-to-domain edges (details in Section 8). Both versions have the same time and space complexity, but the protocols in this paper are much simpler conceptually. Also the view-update protocol is not in reference [1].

Organization of the paper

In Section 2, we present some definitions used in this paper. In Section 3, we define the view data structures. In Section 4, we describe how views are affected by topology changes. In Section 5, we present the view-query protocol. In Section 6, we present the view-update protocol. In Section 7, we present our evaluation model and the results of its application to the superdomain hierarchy. In Section 8, we survey recent approaches to inter-domain routing. In Section 9, we conclude and describe cacheing and heuristic schemes to improve performance.

2 Preliminaries

Each domain has a unique id. Let `DomainIds` denote the set of domain-ids. Each node has a unique id. Let `NodeIds` denote the set of node-ids. For a node x, we use $\mathtt{domainid}(x)$ to denote the domain-id of x's domain.

The superdomain hierarchy defines the following parent-child relationship: a level i, $i > 1$, superdomain is the parent of each level $i-1$ superdomain it contains. Top-level superdomains have no parents. Level 1 superdomains, which are just domains, have no children. For any two superdomains X and Y, X is a sibling of Y iff X and Y have the same parent. X is an ancestor (descendant) of Y iff $X = Y$ or X is an ancestor (descendant) of Y's parent (child).

Each router maintains information about a subset of superdomains, referred to as its visible superdomains. The *visible superdomains* of a router x are (1) x's domain itself, (2) siblings of x's domain, and (3) siblings of ancestors of x's domain. In Figure 3, the visible superdomains of a router in $d1$ are $d1, d2, d3, B, C, G, J$ (these are shown in Figure 4). Note that if a superdomain U is visible to a router, then no ancestor or descendant of U is visible to the router.

Each superdomain has a unique id, i.e. unique among all superdomains regardless of level. Let `SuperDomainIds` denote the set of superdomain-ids. `DomainIds` is a subset of `SuperDomainIds`. For a superdomain U, let `level`(U) denote the level of U in the hierarchy, let `Ancestors`(U) denote the set of ids of ancestor superdomains of U in the hierarchy, and let `Children`(U) denote the set of ids of child superdomains of U in the hierarchy.

For a router x, let `VisibleSuperDomains`(x) denote the set of ids of superdomains visible from x.

We extend the above definitions by allowing their arguments to be nodes, in which case the node stands for its domain. For example, if x is a node in domain d, `Ancestors`(x) denotes `Ancestors`(d).

3 Superdomain-Level Views with Gateways

For routing purposes, each domain (and node) has an address, defined as the concatenation of the superdomain ids starting from the top level and going down to the domain (node). For example in Figure 3, the address of domain $d15$ is $G.E.d15$, and the address of a node h in $d15$ is $G.E.d15.h$.

When a source node needs to reach a destination node, it first determines the visible superdomain in the destination address and then by examining its view determines a superdomain-level source route (satisfying ToS and policy constraints) to this superdomain. However, since routers in different superdomains maintain views of different sets of superdomains, this superdomain-level source route can be meaningless at some intermediate superdomain's router x because the next superdomain in this source route is not visible to x. For example in Figure 4, superdomain-level source route $\langle d2, B, G, C\rangle$ created at a router in $d2$ becomes meaningless once the packet is in G, where C is not visible.

The usual technique of solving this problem is to augment superdomain-level views with gateways and edges between these gateways.

Define the pair $U{:}g$ to be an *sd-gateway* iff U is a superdomain and g is a node that is in U and has a link to a node outside U. Equivalently, we say that g is *a gateway of* U.

Define $\langle U{:}g, h\rangle$ to be an *actual-edge* iff $U{:}g$ is an sd-gateway, h is a gateway not in U, and there is a link from g to h.

Define $\langle U{:}g, h\rangle$ to be a *virtual-edge* iff $U{:}g$ and $U{:}h$ are sd-gateways and $g \neq h$ (note that there may not be a link between g and h).

$\langle U{:}g, h\rangle$ is an *edge* iff it is an actual-edge or a virtual-edge. An edge $\langle U{:}g, h\rangle$ is also said to be an *outgoing edge* of $U{:}g$. Define *edges of* $U{:}g$ to be the set of edges outgoing from $U{:}g$. Define *edges of* U to be the set of edges outgoing from any gateway of U.

Let `Gateways`(U) denote the set of node-ids of gateways of U. Let `Edges`$(U{:}g)$ denote the edges of $U{:}g$. Note that we never use "edge" as a synonym for link.

A gateway g of a domain can generate many sd-gateways, specifically, $U{:}g$ for every ancestor U of g's domain such that g has a link to a node outside U. A link $\langle g, h\rangle$ where g and h are gateways in different domains, can generate many actual-edges; specifically, actual-edge $\langle U{:}g, h\rangle$ for every ancestor U of g's domain such that U is not an ancestor of h's domain.

For the internetwork topology of Figure 2, the corresponding gateway-level connections are shown in Figure 6 where black rectangles are gateways. For the hierarchy of Figure 3, gateway g in Figure 6 generates sd-gateways $d16{:}g$, $E{:}g$, and $G{:}g$. The link $\langle g, h\rangle$ in Figure 6 generates actual-edges $\langle d16{:}g, h\rangle$, $\langle E{:}g, h\rangle$, $\langle G{:}g, h\rangle$.

To a router, at most one of the sd-gateways generated by a gateway g is visible, namely $U{:}g$ where U is an ancestor of g's domain and U is visible to the router. At most one of the actual-edges generated by a link $\langle g, h\rangle$ between two gateways in different domains is visible to the router, namely edge $\langle U{:}g, h\rangle$ where $U{:}g$ is visible to the router. None of the actual-edges are visible to the router if g and h are inside a visible superdomain. For example in Figure 3, of the actual-edges generated by link $\langle g, h\rangle$, only $\langle G{:}g, h\rangle$ is visible to a router in $d1$, and only $\langle d16{:}g, h\rangle$ is visible to a router in $d16$.

A router maintains a view consisting of the visible sd-gateways and their outgoing actual- and virtual-edges. An edge $\langle U{:}g, h\rangle$ in the view of a router connects the sd-gateway $U{:}g$ to the sd-

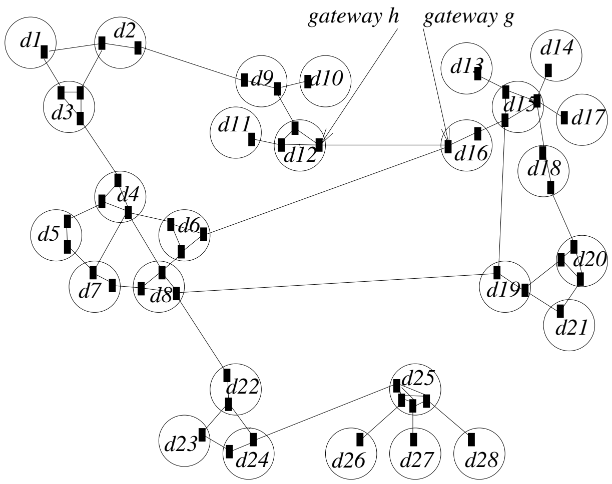

Figure 6: Gateway-level connections of internetwork of Figure 2.

gateway $V{:}h$ such that $V{:}h$ is visible to the router. For the superdomain-level views of Figures 4 and 5, the new views are shown in Figures 7 and 8, respectively.

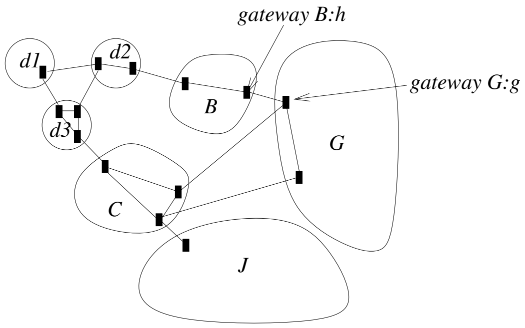

Figure 7: View of a router in $d1$.

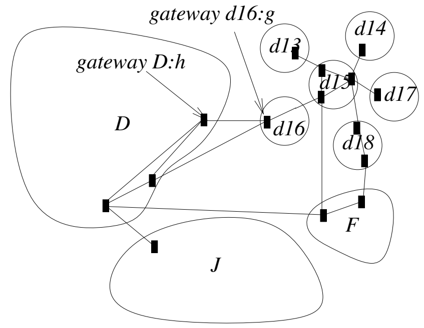

Figure 8: View of a router in $d16$.

The view of a router x contains, for each superdomain U that is visible to x or is an ancestor of x, the strong and weak constraints of U and a set referred to as $\mathit{Gateways\&Edges}_x(U)$. This set contains, for each gateway y of U, the edges of $U{:}y$ and their costs. The reason for storing information about ancestor superdomains is given in Section 5. The cost field is used to satisfy ToS constraints and is described in Section 4. The *timestamp* field is described in Section 6. Formally, the view of x is defined as follows:

$View_x$. View of x.

$$= \{\langle U,\ \texttt{strong\_constraints}(U),\ \texttt{weak\_constraints}(U),\ \mathit{Gateways\&Edges}_x(U)\rangle :$$
$$U \in \texttt{VisibleSuperDomains}(x) \cup \texttt{Ancestors}(x)\ \}$$

where

$\mathit{Gateways\&Edges}_x(U)$. Sd-gateways and edges of U.

$$= \{\langle y,\ \mathit{timestamp},\ \{\langle z,\ \mathit{cost}\rangle :\ \langle U{:}y, z\rangle \in \texttt{Edges}(U{:}y)\}\rangle :\ y \in \texttt{Gateways}(U)\ \}.$$

ToS and policy constraints can also be specified for each sd-gateway and edge. Our protocols can be extended to handle such constraints, but we have not done so here in order to keep their descriptions simple.

A *superdomain-level source route* is now a sequence of sd-gateway ids. With this definition, it is easy to verify that whenever the next superdomain in a superdomain-level source route is not visible to a router, there is an actual-edge (hence a link) between the router and the next gateway in this route.

4 Edge-Costs and Topology Changes

A cost is associated with each edge. The cost of an edge equals a vector of values if the edge is up; each cost value indicates how expensive it is to cross the edge according to some ToS constraint. The cost equals ∞ if the edge is an actual-edge and it is down, or the edge is a virtual-edge $\langle U{:}g, h\rangle$ and h can not be reached from g without leaving U.

Since an actual-edge represents a physical link, its cost can be determined from measured link statistics. The cost of a virtual-edge $\langle U{:}g, h\rangle$ is an aggregation of the cost of physical links in U and is calculated as follows: If U is a domain, the cost of $\langle U{:}g, h\rangle$ is calculated as the maximum/minimum/average cost of the routes within U from g to h [4]. For higher level superdomains U, the cost of $\langle U{:}g, h\rangle$ is derived from the costs of edges between the gateways of children superdomains of U.

Link cost changes and link/node failures and repairs correspond to cost changes, failures and repairs of actual- and virtual-edges. Thus the attributes of edges in the views of routers must be regularly updated. For this, we employ a view-update protocol (see Section 6).

Link/node failures can also partition a superdomain into cells, where a *cell* of a superdomain is defined to be a maximal subset of nodes of the superdomain that can reach each other without leaving the superdomain. Superdomain partitions can occur at any level in the hierarchy. For example, suppose U is a domain and V is its parent superdomain. U can be partitioned into cells without V being partitioned (i.e. if the cells of U can reach each other without leaving V). The opposite can also happen: if all links between U and the other children of V fail, then V becomes partitioned but U does not. Or both U and V can be partitioned. In the same way, link/node repairs can merge cells into bigger cells.

We handle superdomain partitioning as follows: A router detects that a superdomain U is partitioned when a virtual-edge of U in the router's view has cost ∞. When a router forwards a packet to a destination for which the visible superdomain, say U, in the destination address is partitioned into cells, a copy of the packet is sent to each cell by sending a copy of the packet to each gateway of U; the id U in the destination address is "marked" in the packet so that subsequent routers do not create new copies of the packet for U.

5 View-Query Protocol

When a source node wants a superdomain-level source route to a destination, a router in its domain examines its view and searches for a valid path (i.e. superdomain-level source route) using the destination address[6]. We refer to this router as the *source router*. Even though the source router does not know the constraints of the individual domains that are to be crossed in each superdomain, it does know the strong and weak constraints of the superdomains. We refer to a superdomain whose strong constraints are satisfied as a *valid superdomain*. If a superdomain's weak constraints are satisfied but strong constraints are not satisfied, then there may be a valid path through this superdomain. We refer to such a superdomain as a *candidate superdomain*.

A path is valid if it involves only valid superdomains. A path cannot be valid if it involves a superdomain which is neither valid nor candidate. We refer to a path involving only valid and candidate superdomains as a *candidate path*.

[6] We assume that the source has the destination's address. If that is not the case, it would first query the name servers to obtain the address for the destination. Querying the name servers can be done the same way it is done currently in the Internet. It requires nodes to have a set of fixed addresses to name servers. This is also sufficient in our case.

If the source router's view contains a candidate path $\langle U_0{:}g_{0_0}, \ldots, U_0{:}g_{0_{n_0}}, U_1{:}g_{1_0}, \ldots, U_1{:}g_{1_{n_1}}, \cdots, U_m{:}g_{m_0}, \ldots, U_m{:}g_{m_{n_m}} \rangle$ to the destination (and does not contain a valid path), then for each candidate superdomain U_i on this path, the source router queries gateway g_{i_0} of U_i for the internal view of U_i. This internal view consists of the constraints, sd-gateways and edges of the child superdomains of U_i.

When a router x receives a request for the internal view of an ancestor superdomain U, it returns the following data structure:

> $IView_x(U)$. Internal view of U at router x.
>
> $= \{\langle V, \texttt{strong\_constraints}(V), \texttt{weak\_constraints}(V), \ Gateways\&Edges_x(V)\rangle \in View_x :$
> $V \in \texttt{Children}(U)\}$

It is to simplify the construction of $IView_x(U)$ that we store information about ancestor superdomains in the view of router x. Instead of storing this information, router x could construct $IView_x(U)$ from the constraints, sd-gateways and edges of the visible descendants of U. We did not choose this alternative because the extra information does not increase storage complexity.

When the source router receives the internal view of a superdomain U, it does the following: (1) it removes the sd-gateways and edges of U from its view; (2) it adds the sd-gateways and edges of children superdomains in the internal view of U; and (3) searches for a valid path again. If there is still no valid path but there are candidate paths, the process is repeated.

For example, consider Figure 3. For a router in superdomain $d1$ (see Figure 7), G is visible and is a candidate domain. The internal view of G is shown in Figure 9, and the resulting merged view is shown in Figure 10. The valid path through G (visiting $d16$ and avoiding $d19$) can be discovered using this merged view (since the strong constraints of E are satisfied).

Consider a candidate route to a destination: $\langle U_0{:}g_{0_0}, \ldots, U_0{:}g_{0_{n_0}}, U_1{:}g_{1_0}, \ldots, U_1{:}g_{1_{n_1}}, \cdots, U_m{:}g_{m_0}, \ldots, U_m{:}g_{m_{n_m}} \rangle$. If superdomain U_i is partitioned into cells, it may re-appear later in the candidate path (i.e. for some $j \neq i$, $U_j = U_i$). In this case both gateways g_{i_0} and g_{j_0} are queried. Timestamps are used to resolve conflicts between the information reported by these gateways.

The view-query protocol uses two types of messages as follows:

- $(\texttt{RequestIView}, sdid, gid, s\_address, d\_address)$

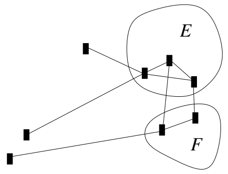

Figure 9: Internal view of G.

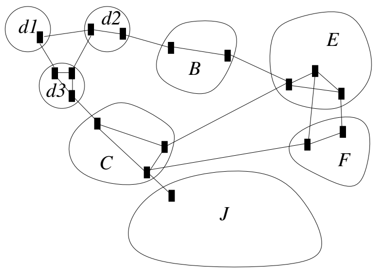

Figure 10: Merged view at $d1$.

Sent by a source router to gateway gid to obtain the internal view of superdomain $sdid$. $s\_address$ is the address of the source router. $d\_address$ is the address of the destination node (of the desired route).

- ($\texttt{ReplyIView}, sdid, gid, iview, d\_address$)
  where $iview$ is the internal view of superdomain $sdid$, and other parameters are as in the `RequestIView` message. It is sent by gateway gid to the source router.

The state maintained by a source router x is listed in Figure 15. $PendingReq_x$ is used to avoid sending new request messages before receiving all outstanding reply messages. $WView_x$ and $PendingReq_x$ are allocated and deallocated on demand for each destination.

The events of router x are specified in Figure 15. In the figure, * is a wild-card matching any value. $TimeOut_x$ event is executed after a time-out period from the execution of $Request_x$ event to indicate that the request has not been satisfied. The source host can then repeat the same request afterwards.

The procedure $search_x$ uses an operation "ReliableSend(m) to v", where m is the message being sent and v is either an address of an arbitrary router or an id of a gateway of a visible superdomain. ReliableSend is asynchronous. The message is delivered to v as long as there is a sequence of up links between u and v.[7] (Note that an address is not needed to obtain an inter-domain route to a gateway of a visible superdomain.)

Router Failure Model: A router can undergo failures and recoveries at anytime. We assume failures are fail-stop (i.e. a failed router does not send erroneous messages). When a router x recovers, the variables $WView_x$ and $PendingReq_x$ are lost for all destinations. The cost of each edge in $View_x$ is set to ∞. It becomes up-to-date as the router receives new information from other

[7] This involves time-outs, retransmissions, etc. It requires a transport protocol support such as TCP.

routers.

6 View-Update Protocol

A gateway g, for each ancestor superdomain U, informs other routers of topology changes (i.e. failures, repairs and cost changes) affecting $U{:}g$'s edges. The communication is done by flooding messages. The flooding is restricted to the routers in the parent superdomain of U, since U is visible only to these routers.

Due to the nature of flooding, a router can receive information out of order from a gateway. In order to avoid old information replacing new information, each gateway includes increasing time stamps in the messages it sends. Routers maintain for each gateway the highest received time stamp (in the *timestamp* field in $View_x$), and discard messages with smaller timestamps. Time stamps do not have to be real-time clock values.

Due to superdomain partitioning, messages sent by a gateway may not reach all routers within the parent superdomain, resulting in some routers having out-of-date information. This out-of-date information can cause inconsistencies when the partition is repaired. To eliminate inconsistencies, when a link recovers, the two routers at the ends of the link exchange their views and flood any new information. As usual, information about a superdomain U is flooded over U's parent superdomain.

The view-update protocol uses messages of the following form:

- (`Update`, *sdid*, *gid*, *timestamp*, *edge-set*)

  Sent by the gateway *gid* to inform other routers about current attributes of edges of *sdid*:*gid*. *timestamp* indicates the time stamp of *gid*. *edge-set* contains a cost for each edge.

The state maintained by a router x is listed in Figure 16. Note that $\texttt{AdjLocalRouters}_x$ or $\texttt{AdjForeignGateways}_x$ can be empty. $IntraDomainRT_x$ contains a route (next-hop or source)[8] for every reachable node of the domain. We assume that consecutive reads of $Clock_x$ returns increasing values.

Routers also receive and flood messages containing edges of sd-gateways of their ancestor superdomains. This information is used by the query protocol (see Section 5). Also the highest timestamp received from a gateway g of an ancestor superdomain is needed to avoid exchanging

[8] $IntraDomainRT_x$ is a view in case of a link-state routing protocol or a distance table in case of a distance-vector routing protocol.

the messages of g infinitely during flooding.

The events of router x are specified in Figure 16. We use $\texttt{Ancestor}_i(U)$ to denote the superdomain-id of the ith ancestor of U, where $\texttt{Ancestor}_0(U) = U$. In the view-update protocol, a node u uses send operations of the form "Send(m) to v", where m is the message being sent and v is the destination-id. Here, nodes u and v are neighbors, and the message is sent over the physical link $\langle u, v\rangle$. If the link is down, we assume that the packet is dropped.

7 Evaluation

In the superdomain hierarchy (without the query protocol), the number of superdomains in a view is logarithmic in the number of superdomains in the internetwork [10].[9] However, the storage required for a view is proportional not to the number of superdomains in it but to the number of sd-gateways in it. As we have seen, there can be more than one sd-gateway for a superdomain in a view.

In fact, the superdomain hierarchy does not scale-up for arbitrary internetworks; that is, the number of sd-gateways in a view can be proportional to the number of domains in the internetwork. For example, if each domain in a superdomain U has a distinct gateway with a link to outside U, the number of sd-gateways of U would be linear in the number of domains in U.

The good news is that the superdomain hierarchy does scale-up for realistic internetwork topologies. A sufficient condition for scaling is that each superdomain has at most $\log N_D$ sd-gateways; this condition is satisfied by realistic internetworks since most domain interconnections are "hierarchical connections" i.e. between backbones and regionals, between regionals and MANs, and so on.

In this section, we present an evaluation of the scaling properties of the superdomain hierarchy and the query protocol. To evaluate any inter-domain routing protocol, we need a model in which we can define internetwork topologies, policy/ToS constraints, inter-domain routing hierarchies, and evaluation measures (e.g. memory and time requirements). We have recently developed such a model [3]. We first describe our model, and then use it to evaluate our superdomain hierarchy. Our evaluation measures are the amount of memory required at the routers, and the amount of

[9] Even though the results in [10] were for intra-domain routing, it is easy to show that the analysis there holds for inter-domain routing as well.

time needed to construct a path.

7.1 Evaluation Model

We first describe our method of generating topologies and policy/ToS constraints. We then describe the evaluation measures.

Generating Internetwork Topologies

For our purposes, an internetwork *topology* is a directed graph where the nodes correspond to domains and the edges correspond to domain-level connections. However, an arbitrary graph will not do. The topology should have the characteristics of a real internetwork, like the Internet. That is, it should have backbones, regionals, MANS, LANS, etc.; there should be hierarchical connections, but some "non-hierarchical" connections should also be present.

For brevity, we refer to backbones as class 0 domains, regionals as class 1 domains, metropolitan-area domains and providers as class 2 domains, and campus and local-area domains as class 3 domains. A (strictly) hierarchical interconnection of domains means that class 0 domains are connected to each other, and for $i > 0$, class i domains are connected to class $i-1$ domains. As mentioned above, we also want some "non-hierarchical" connections, i.e., domain-level edges between domains irrespective of their classes (e.g. from a campus domain to another campus domain or to a backbone domain).

In reality, domains span geographical regions and domain-level edges are often between domains that are geographically close (e.g. University of Maryland campus domain is connected to SURANET regional domain which are both in the east coast). We also want some edges that are between far domains. A class i domain usually spans a larger geographical region than a class $i+1$ domain. To generate such interconnections, we associate a "region" attribute to each domain. The intention is that two domains with the same region are geographically close.

The *region* of a class i domain has the form $\mathtt{r_0.r_1.\cdots.r_i}$, where the r_j's are integers. For example, the region of a class 3 domain can be `1.2.3.4`. For brevity, we refer to the region of a class i domain as a class i region.

Note that regions have their own hierarchy which should not be confused with the superdomain hierarchy. Class 0 regions are the top level regions. We say that a class i region $\mathtt{r_0.r_1.\cdots.r_{i-1}.r_i}$

is *contained* in the class $i-1$ region $\texttt{r}_0.\texttt{r}_1.\cdots.\texttt{r}_{i-1}$ (where $i > 0$). Containment is transitive. Thus region `1.2.3.4` is contained in regions `1.2.3`, `1.2` and `1`.

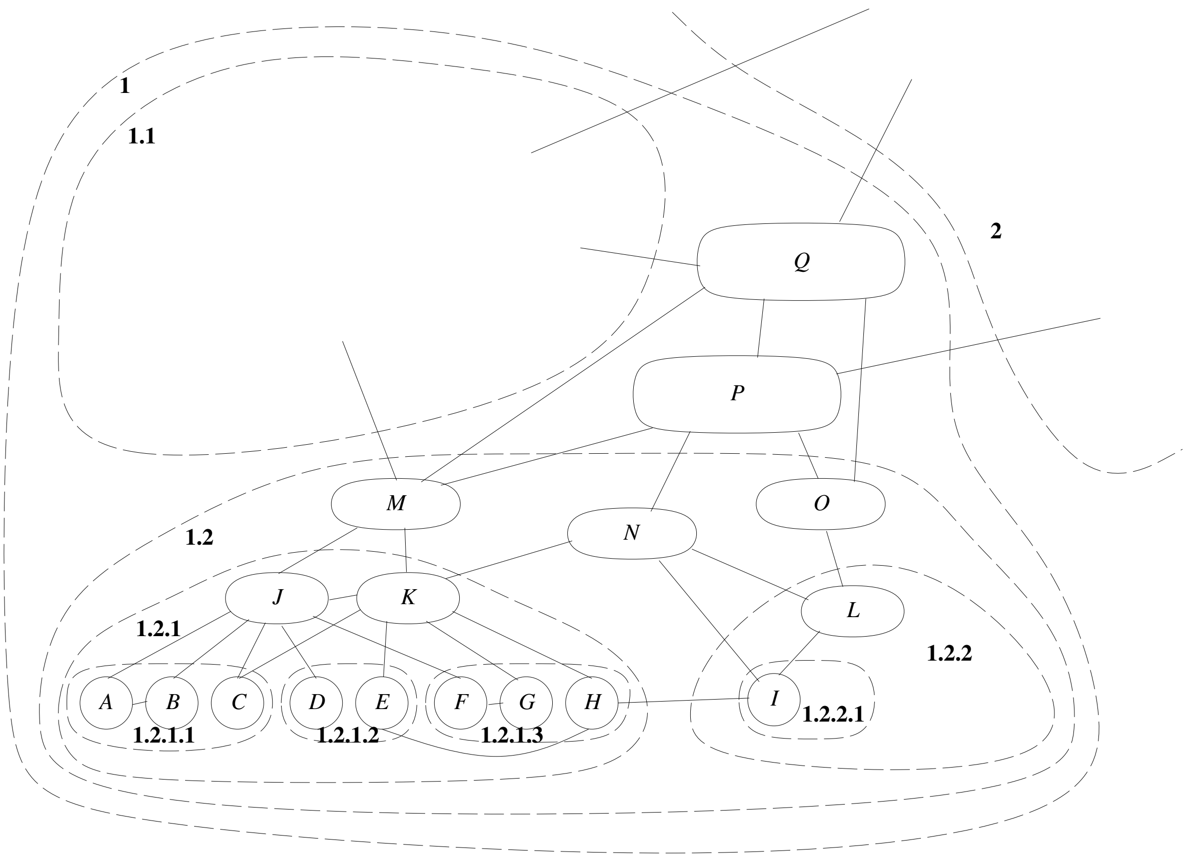

Figure 11: Regions

Given any pair of domains, we classify them as local, remote or far, based on their regions. Let X be a class i domain and Y a class j domain, and without loss of generality let $i \leq j$. X and Y are *local* if they are in the same class i region. For example in Figure 11, A is local to B, C, J, K, M, N, O, P, and Q. X and Y are *remote* if they are not in the same class i region but they are in the same class $i-1$ region, or if $i = 0$. For example in Figure 11, some of the domains A is remote to are D, E, F, and L. X and Y are *far* if they are not local or remote. For example in Figure 11, A is far to I.

We refer to a domain-level edge as *local* (*remote*, or *far*) if the two domains it connects are local (remote, or far).

We use the following procedure to generate internetwork topologies:

- We first specify the number of domain classes, and the number of domains in each class.
- We next specify the regions. Note that the number of region classes equals the number of domain classes. We specify the number of class 0 regions. For each class $i > 0$, we specify a *branching factor*, which creates that many class i regions in each class $i-1$ region. (That is, if there are two class 0 regions and the class 1 branching factor equals three, then there are six class 1 regions.)
- For each class i, we randomly map the class i domains into the class i regions. Note that several domains can be mapped to the same region, and some regions may have no domain mapped into them.
- For every class i and every class j, $j \geq i$, we specify the number of local, remote and far edges to be introduced between class i domains and class j domains. The end points of the edges are chosen randomly (within the specified constraints).
- We ensure that the internetwork topology is connected by ensuring that the subgraph of class 0 domains is connected, and each class i domain, for $i > 0$, is connected to a local class $i-1$ domain.
- Each domain has one gateway. So all neighbors of a domain are connected via this gateway. This is for simplicity.

Choosing Policy/ToS Constraints

We chose a simple scheme to model policy/ToS constraints. Each domain is assigned a color: *green* or *red*. For each domain class, we specify the percentage of green domains in that class, and then randomly choose a color for each domain in that class.

A *valid route* from a source to a destination is one that does not visit any red intermediate domains; the source and destination domains are allowed to be red.

This simple scheme can model many realistic policy/ToS constraints, such as security constraints and bandwidth requirements. It cannot model some important kinds of constraints, such as delay bounds.

Computing Evaluation Measures

The evaluation measures of most interest for an inter-domain routing protocol are its memory, time and communication requirements. We postpone the precise definitions of the evaluation measures to the next subsection.

The only analysis method we have at present is to numerically compute the evaluation measures for a variety of source-destination pairs. Because we use internetwork topologies of large sizes, it is not feasible to compute for all possible source-destination pairs. We randomly choose a set of source-destination pairs that satisfy the following conditions: (1) the source and destination domains are different stub domains, and (2) there exists a valid path from the source domain to the destination domain in the internetwork topology. (Note that the straight-forward scheme would always find such a path.)

7.2 Application to Superdomain Query Protocol

We use the above model to evaluate our superdomain query protocol for several different superdomain hierarchies. For each hierarchy, we define a set of superdomain-ids and a parent-child relationship on them.

The first superdomain hierarchy scheme is referred to as *child-domains*. Each domain d (regardless of its class) is a level-1 superdomain, also identified as d. In addition, for each backbone d, we create a distinct level-4 superdomain referred to as d-4. For each regional d, we create a distinct level-3 superdomain d-3 and make it a child of a randomly chosen level-4 superdomain e-4 such that d and e are local and connected. For each MAN d, we create a distinct level-2 superdomain d-2 and make it a child of a randomly chosen level-3 superdomain e-3 such that d and e are local and connected. Please see Figure 12.

We next describe how the level-1 superdomains (i.e. the domains) are placed in the hierarchy. A backbone d is placed in, i.e. as a child of, d-4. A regional d is placed in d-3. A MAN d is placed in d-2. A stub d is placed in e-2 such that d and e are local and connected. Please see Figure 12.

The second superdomain hierarchy scheme is referred to as *sibling-domains*. It is identical to *child-domains* except for the placement of level-1 superdomains corresponding to backbones, regionals and MANs. In *sibling-domains*, a backbone d is placed as a sibling of d-4. A regional d is placed as a sibling of d-3. A MAN d is placed as a sibling of d-2. Please see Figure 13.

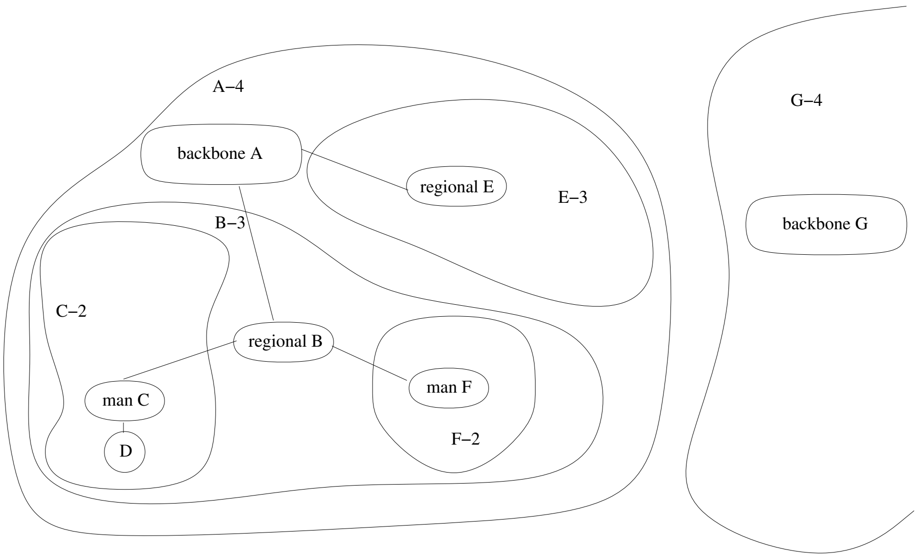

Figure 12: *child-domains*

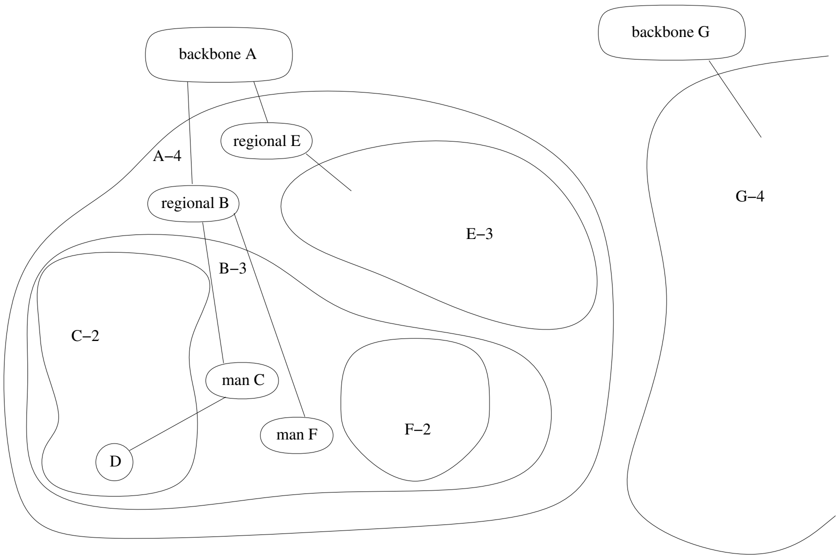

Figure 13: *sibling-domains*

The third superdomain hierarchy scheme is referred to as *leaf-domains*. It is identical to *child-domains* except for the placement of level-1 superdomains corresponding to backbones and regionals.

In *leaf-domains*, backbones and regionals are placed in some level-2 superdomain, as follows. A regional d, if superdomain d-3 has a child superdomain e-2, is placed in e-2. Otherwise, a new level-2 superdomain d-2 is created and placed in d-3. d is placed in d-2. A backbone d, if superdomain d-4 has a child superdomain f-3, is placed in the level-2 superdomain containing the regional f. Otherwise, a new level-3 superdomain d-3 is created and placed in d-4, a new level-2 superdomain d-2 is created and placed in d-3. d is placed in d-2. Please see Figure 14.

Note that in *leaf-domains*, all level-1 superdomains are placed under level-2 superdomains. Whereas other schemes allow some level-1 superdomains to be placed under higher level superdomains.

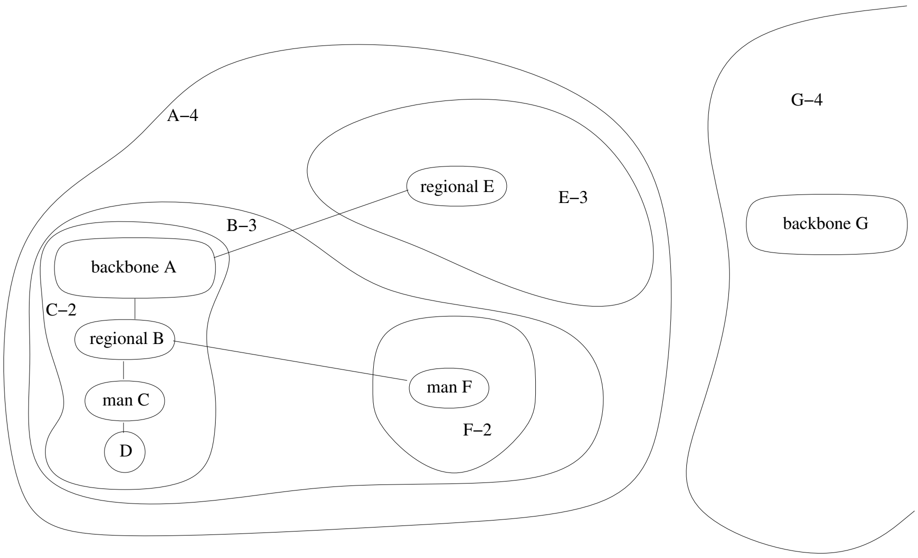

Figure 14: *leaf-domains*

The fourth superdomain hierarchy scheme is referred to as *regions*. In this scheme, the superdomain hierarchy corresponds exactly to the region hierarchy used to generate the internetwork topology. That is, for a class 1 region `x` there is a distinct level 5 (top level) superdomain x-5. For a class 2 region `x.y` there is a distinct level 4 superdomain $x.y$-4 placed under level 5 superdomain x-5, and so on. Each domain is placed under the superdomain of its region. Please see Figure 11.

Results for Internetwork 1

The parameters of the first internetwork topology, referred to as Internetwork 1, are shown in Table 1.

Class i	No. of Domains	No. of Regions[10]	% of Green Domains	Edges between Classes i and j			
				Class j	Local	Remote	Far
0	10	4	0.80	0	8	6	0
1	100	16	0.75	0	190	20	0
				1	26	5	0
2	1000	64	0.70	0	100	0	0
				1	1060	40	0
				2	200	40	0
3	10000	256	0.20	0	100	0	0
				1	100	0	0
				2	10100	50	0
				3	50	50	50

Table 1: Parameters of Internetwork 1.

Our evaluation measures were computed for a (randomly chosen but fixed) set of 100,000 source-destination pairs. For a source-destination pair, we refer to the length of the shortest valid path in the internetwork topology as the *shortest-path length*, or *spl* in short. The minimum *spl* of these pairs was 2, the maximum *spl* was 15, and the average *spl* was 6.84.

For each source-destination pair, the set of candidate paths is examined in shortest-first order until either a valid path was found or the set was exhausted and no valid paths were found. For each candidate path, `RequestIView` messages are sent to all candidate superdomains on this path in parallel. All `ReplyIView` messages are received in time proportional to the round-trip time to the farthest of these superdomains. Hence, total time requirement is proportional to the number of candidate paths queried multiplied by the round-trip time to the farthest superdomain in these paths. Let *msgsize* denote the sum of average `RequestIView` message size and average

[10] Branching factor is 4 for all region classes.

Scheme	No query needed	Candidate Paths	Candidate Superdomains
child-domains	220	3.31/13	7.35/38
sibling-domains	220	3/10	6.17/22
leaf-domains	219	6.31/24	15.94/66
regions	544	3.70/12	7.79/30

Table 2: Queries for Internetwork 1.

`ReplyIView` message size. The number of candidate superdomains queried times *msgsize* indicates the communication capacity required to ship the `RequestIView` and `ReplyIView` messages.

Table 2 lists for each superdomain scheme the average and maximum number of candidate paths and candidate superdomains queried. As apparent from the table, *sibling-domains* is superior to other schemes and *leaf-domains* is much worse than the rest. This is because in *leaf-domains*, even if only one domain d in a superdomain U is actually going to be crossed, all descendants of U containing d may need to be queried to obtain a valid path (e.g. to cross backbone A in Figure 14, it may be necessary to query for superdomain A-4, then B-3, then C-2).

	Initial view size		Merged view size	
Scheme	in sd-gateways	in superdomains	in sd-gateways	in superdomains
child-domains	964/1006	42/60	1089/1282	100/298
sibling-domains	1167/1269	70/99	1470/2190	148/337
leaf-domains	963/1006	40/60	1108/1322	130/411
regions	492/715	85/163	1042/2687	158/369

Table 3: View sizes for Internetwork 1.

Table 3 lists for each superdomain scheme the average and maximum of the initial view size and of the merged view size. The initial view size indicates the memory requirement at a router without using the query protocol (i.e. assuming the initial view has a valid path). The merged view size indicates the memory requirement at a router during the query protocol (after finding a valid path). The memory requirement at a router is O(view size in number of sd-gateways $\times E_G$) where E_G is the average number of edges of an sd-gateway. Note that the source does not need to store information about red and non-transit domains in the merged views (other than the ones already in the initial view). The numbers for the merged view sizes in Table 3 take advantage of this.

As apparent from the table, *leaf-domains*, *child-domains* and *regions* scale better than *sibling-domains*. There are two reasons for this. First, placing a backbone (regional or MAN) domain d as a sibling to d-4 (d-3 or d-2) doubles the number of level 4 (3 or 2) superdomains in the views of routers. Second, since these domains have many edges to the domains in their associated superdomains, the end points of each of these edges become sd-gateways of the associated superdomains. Note that *regions* scales much superior to the other schemes in the initial view size. This is because most edges are local (i.e. contained within regions), thus contained completely in superdomains. Hence, their end points are not sd-gateways.

Overall, the *child-domains* and *regions* schemes scale best in space, time and communication requirements. We have repeated the above evaluations for two other internetworks and obtained similar conclusions. The results are in Appendix A.

8 Related Work

In this section, we survey recently proposed inter-domain routing protocols that support ToS and policy routing for large internetworks.

Nimrod [6] and IDPR [16] use the link-state approach with domain-level source routing to enforce policy and ToS constraints and superdomains to solve scaling problem. Nimrod is still in a design stage. Both protocols suffer from loss of policy and ToS information as mentioned in the introduction. A query protocol for Nimrod is being developed to obtain more detailed policy, ToS and topology information.

BGP [12] and IDRP [14] are based on a *path-vector* approach [15]. Here, for each destination domain a router maintains a set of paths, one through each of its neighbor routers. ToS and policy information is attached to these paths. Each router requires $O(N_D \times N_D \times E_R)$ space, where N_D is the average number of neighbor domains for a domain and N_R is the number of routers in the internetwork. For each destination, a router exchanges its best valid path with its neighbor routers. However, a path-vector algorithm may not find a valid path from a source to the destination even if such a route exists [16][11] (i.e. detailed ToS and policy information may be lost). By exchanging k paths to each destination, the probability of detecting a valid path for each source can be increased. But to guarantee detection, either all possible paths should be exchanged (exponential number of paths in the worst case) or source policies should be made public and routers should take this into account when exchanging routes. However, this fix increases space and communication requirements drastically.

IDRP [14] uses superdomains to solve the scaling problem. It exchanges all paths between neighbor routers subject to the following constraint: a router does not inform a neighbor router of a route if usage of the route by the neighbor would violate some superdomain's constraint on the route. IDRP also suffers from loss of ToS and policy information. To overcome this problem, it uses overlapping superdomains: that is, a domain and superdomain can be in more than one parent superdomain. If a valid path over a domain can not be discovered because the constraints of a parent superdomain are violated, the same path may be discovered through another parent superdomain whose constraints are not violated. However, handling ToS and policy constraints in general requires more and more combinations of overlapping superdomains, resulting in more storage requirement.

Reference [9] combines the benefits of path-vector approach and link-state approach by having two modes: An NR mode, which is an extension of IDRP and is used for the most common ToS and policy constraints; and a SDR mode, which is like IDPR and is used for less frequent ToS and policy requests. This study does not address the scalability of the SDR mode. Ongoing work by this group considers a new SDR mode which is not based on IDPR.

Reference [19] suggests the use of multiple addresses for each node, one for each ToS and Policy. This scheme does not scale up. In fact, it increases the storage requirement, since a router maintains a route for each destination address, and there are more addresses with this scheme.

The landmark hierarchy [18, 17] is another approach for solving scaling problem. Here, each router is a landmark with a radius, and routers which are at most radius away from the landmark maintain a route for it. Landmarks are organized hierarchically, such that radius of a landmark increases with its level, and the radii of top level landmarks include all routers. Addressing and

[11] For example, suppose a router u has two paths $P1$ and $P2$ to the destination. Let u have a router neighbor v, which is in another domain. u chooses and informs v of one of the paths, say $P1$. But $P1$ may violate source policies of v's domain, and $P2$ may be a valid path for v.

packet forwarding schemes are introduced. Link-state algorithms can not be used with the landmark hierarchy, and a thorough study of enforcing ToS and policy constraints with this hierarchy has not been done.

In [1], we provided an alternative solution to loss of policy and ToS information that is perhaps more faithful to the original superdomain hierarchy. To handle superdomain-level source routing and topology changes, we augmented each superdomain-level edge (U, V) with the address of an "exit" domain u in U and an "entry" domain v in V. To obtain internal views, we added for each visible superdomain U the edges from U to domains outside the parent of U. Surprisingly, this approach and the gateway-level view approach have the same memory and communication requirements. However, the first approach results in much more complicated protocols.

Reference [2] presents interdomain routing protocols based on a new kind of hierarchy, referred to as the viewserver hierarchy. This approach also scales well to large internetworks and does not lose detail ToS and policy information. Here, special routers called viewservers maintain the view of domains in a surrounding precinct. Viewservers are organized hierarchically such that for each viewserver, there is a domain of a lower level viewserver in its view, and views of top level viewservers include domains of other top level viewservers. Appropriate addressing and route discovery schemes are introduced.

9 Conclusion

We presented a hierarchical inter-domain routing protocol which satisfies policy and ToS constraints, adapts to dynamic topology changes including failures that partition domains, and scales well to large number of domains.

Our protocol achieves scaling in space requirement by using superdomains. Our protocol maintains superdomain-level views with sd-gateways and handles topology changes by using a link-state view update protocol. It achieves scaling in communication requirement by flooding topology changes affecting a superdomain U over U's parent superdomain.

Our protocol does not lose detail in ToS, policy and topology information. It stores both a strong set of constraints and a weak set of constraints for each visible superdomain. If the weak constraints but not the strong constraints of a superdomain U are satisfied (i.e. the aggregation has resulted in loss of detail in ToS and policy information), then some paths through U may be valid.

Our protocol uses a query protocol to obtain a more detailed "internal" view of such superdomains, and searches again for a valid path. Our evaluation results indicate that the query protocol can be performed using 15% extra space.

One drawback of our protocols is that to obtain a source route, views are merged at or prior to the connection setup, thereby increasing the setup time. This drawback is not unique to our scheme [7, 16, 6, 9]. There are several ways to reduce this setup overhead. First, source routes to frequently used destinations can be cached. Second, the internal views of frequently queried superdomains can be cached at routers close to the source domain. Third, better heuristics to choose candidate paths and candidate superdomains to query can be developed.

We also described an evaluation model for inter-domain routing protocols. This model can be applied to other inter-domain routing protocols. We have not done so because precise definitions of the hierarchies in these protocols are not available. For example, to do a fair evaluation of IDPR[16], we need precise guidelines for how to group domains into superdomains, and how to choose between the strong and weak methods when defining policy/ToS constraints of superdomains. In fact, these protocols have not been evaluated in a way that we can compare them to the superdomain hierarchy.

References

[1] C. Alaettinoğlu and A. U. Shankar. Hierarchical Inter-Domain Routing Protocol with On-Demand ToS and Poicy Resolution. In Proc. *IEEE International Conference on Networking Protocols '93*, San Fransisco, California, October 1993.

[2] C. Alaettinoğlu and A. U. Shankar. Viewserver Hierarchy: A New Inter-Domain Routing Protocol and its Evaluation. Technical Report UMIACS-TR-93-98, CS-TR-3151, Department of Computer Science, University of Maryland, College Park, October 1993. Earlier version CS-TR-3033, February 1993.

[3] C. Alaettinoğlu and A. U. Shankar. Viewserver Hierarchy: A New Inter-Domain Routing Protocol. In Proc. *IEEE INFOCOM '94*, Toronto, Canada, June 1994. To appear.

[4] A. Bar-Noy and M. Gopal. Topology Distribution Cost vs. Efficient Routing in Large Networks. In Proc. *ACM SIGCOMM '90*, pages 242–252, Philadelphia, Pennsylvania, September 1990.

[5] L. Breslau and D. Estrin. Design of Inter–Administrative Domain Routing Protocols. In Proc. *ACM SIGCOMM '90*, pages 231–241, Philadelphia, Pennsylvania, September 1990.

[6] I. Castineyra, J. N. Chiappa, C. Lynn, R. Ramanathan, and M. Steenstrup. The Nimrod Routing Architecture. Internet Draft., March 1994. Available by anonymous ftp from `research.ftp.com:pub/nimrod`.

[7] D.D. Clark. Policy routing in Internet protocols. Request for Comment RFC-1102, Network Information Center, May 1989.

[8] D. Estrin. Policy requirements for inter Administrative Domain routing. Request for Comment RFC-1125, Network Information Center, November 1989.

[9] D. Estrin, Y. Rekhter, and S. Hotz. Scalable Inter-Domain Routing Architecture. In Proc. *ACM SIGCOMM '92*, pages 40–52, Baltimore, Maryland, August 1992.

[10] L. Kleinrock and F. Kamoun. Hierarchical Routing for Large Networks. *Computer Networks and ISDN Systems*, (1):155–174, 1977.

[11] B.M. Leiner. Policy issues in interconnecting networks. Request for Comment RFC-1124, Network Information Center, September 1989.

[12] K. Lougheed and Y. Rekhter. Border Gateway Protocol (BGP). Request for Comment RFC-1105, Network Information Center, June 1989.

[13] R. Perlman. Hierarchical Networks and Subnetwork Partition Problem. *Computer Networks and ISDN Systems*, 9:297–303, 1985.

[14] Y. Rekhter. Inter-Domain Routing Protocol (IDRP). Available from the author., 1992. T.J. Watson Research Center, IBM Corp.

[15] K. G. Shin and M. Chen. Performance Analysis of Distributed Routing Strategies Free of Ping-Pong-Type Looping. *IEEE Transactions on Computers*, 1987.

[16] M. Steenstrup. An Architecture for Inter-Domain Policy Routing. Request for Comment RFC-1478, Network Information Center, July 1993.

[17] P. F. Tsuchiya. The Landmark Hierarchy: Description and Analysis, The Landmark Routing: Architecture Algorithms and Issues. Technical Report MTR-87W00152, MTR-87W00174, The MITRE Corporation, McLean, Virginia, 1987.

[18] P. F. Tsuchiya. The Landmark Hierarchy:A New Hierarchy For Routing In Very Large Networks. In Proc. *ACM SIGCOMM '88*, August 1988.

[19] P. F. Tsuchiya. Efficient and Robust Policy Routing Using Multiple Hierarchical Addresses. In Proc. *ACM SIGCOMM '91*, pages 53–65, Zurich, Switzerland, September 1991.

A Results for Other Internetworks

Results for Internetwork 2

The parameters of the second internetwork topology, referred to as Internetwork 2, are the same as the parameters of Internetwork 1 but a different seed is used for the random number generation.

Our evaluation measures were computed for a set of 100,000 source-destination pairs. The minimum *spl* of these pairs was 1, the maximum *spl* was 14, and the average *spl* was 7.13.

Table 5 and Table 4 shows the results. Similar conclusions as in the case of Internetwork 1 hold.

Results for Internetwork 3

The parameters of the third internetwork topology, referred to as Internetwork 3, are shown in Table 6. Internetwork 3 is more connected, more class 0, 1 and 2 domains are green, and more class 3 domains are red. Hence, we expect bigger view sizes in number of sd-gateways.

Scheme	No query needed	Candidate Paths	Candidate Superdomains
child-domains	205	4.52/20	10.22/47
sibling-domains	205	3.01/8	6.50/21
leaf-domains	205	8.80/32	21.34/82
regions	640	3.52/10	7.85/28

Table 4: Queries for Internetwork 2.

	Initial view size		Merged view size	
Scheme	in sd-gateways	in superdomains	in sd-gateways	in superdomains
child-domains	958/1012	43/60	1079/1269	118/306
sibling-domains	1153/1283	72/101	1480/2169	160/324
leaf-domains	956/1009	41/58	1095/1281	156/387
regions	624/1024	110/231	1356/3578	206/435

Table 5: View sizes for Internetwork 2.

Our evaluation measures were computed for a set of 100,000 source-destination pairs. The minimum *spl* of these pairs was 1, the maximum *spl* was 11, and the average *spl* was 5.95.

Table 8 and Table 7 shows the results. Similar conclusions as in the cases of Internetwork 1 and 2 hold.

[12] Branching factor is 4 for all domain classes.

Class i	No. of Domains	No. of Regions[12]	% of Green Domains	Edges between Classes i and j			
				Class j	Local	Remote	Far
0	10	4	0.85	0	8	7	0
1	100	16	0.80	0	190	20	0
				1	50	20	0
2	1000	64	0.75	0	500	50	0
				1	1200	100	0
				2	200	40	0
3	10000	256	0.10	0	300	50	0
				1	250	100	0
				2	10250	150	50
				3	200	150	100

Table 6: Parameters of Internetwork 3.

Scheme	No query needed	Candidate Paths	Candidate Superdomains
child-domains	142	3.99/29	7.70/43
sibling-domains	142	2.95/10	5.39/22
leaf-domains	142	9.65/70	18.99/103
regions	676	3.47/17	6.25/21

Table 7: Queries for Internetwork 3.

Scheme	Initial view size in sd-gateways	Initial view size in superdomains	Merged view size in sd-gateways	Merged view size in superdomains
child-domains	2160/2239	43/60	2354/2647	107/348
sibling-domains	2365/2504	72/101	2606/3314	148/356
leaf-domains	2159/2236	41/58	2386/2645	160/648
regions	1107/1644	110/231	1850/3559	194/436

Table 8: View sizes for Internetwork 3.

Variables:

$View_x$. Dynamic view of x.

$WView_x(d\_address)$. Temporary view of x. $d\_address$ is the destination address.
Used for merging internal views of superdomains to the view of x.

$PendingReq_x(d\_address)$. Integer. $d\_address$ is the destination address.
Number of outstanding request messages.

Events:

$Request_x(d\_address)$ {Executed when x wants a valid domain-level source route}
allocate $WView_x(d\_address) := View_x$; allocate $PendingReq_x(d\_address) := 0$;
$search_x(d\_address)$;

where
$search_x(d\_address)$
if there is a valid path to $d\_address$ in $WView_x(d\_address)$ then
$result$:= shortest valid path;
deallocate $WView_x(d\_address)$, $PendingReq_x(d\_address)$;
return $result$;
else if there is a candidate path to $d\_address$ in $WView_x(d\_address)$ then
Let $cpath = \langle U_0{:}g_{0_0}, \ldots, U_0{:}g_{0_{n_0}}, U_1{:}g_{1_0}, \ldots, U_1{:}g_{1_{n_1}}, \cdots, U_m{:}g_{m_0}, \ldots, U_m{:}g_{m_{n_m}} \rangle$
be the shortest candidate path;
for U_i in $cpath$ such that U_i is candidate do
ReliableSend(`RequestIView`, $U_i, g_{i_0}, address(x), d\_address$) to g_{i_0}
$PendingReq_x(d\_address) := PendingReq_x(d\_address) + 1$;
else
deallocate $WView_x(d\_address)$, $PendingReq_x(d\_address)$;
return failure;
endif
endif

$TimeOut_x(d\_address)$ {Executed after a time-out period and $PendingReq_x(d\_address) \neq 0$.}
deallocate $WView_x(d\_address)$, $PendingReq_x(d\_address)$;
return failure;

Figure 15: view-query protocol: State and events of a router x. (Figure continued on next page.)

```
Receive_x(RequestIView, sdid, x, s_address, d_address)
    ReliableSend(ReplyIView, sdid, x, IView_x(U), d_address) to s_address;

Receive_x(ReplyIView, sdid, gid, iview, d_address)
    if PendingReq_x(d_address) ≠ 0 then            {No time-out happened}
        PendingReq_x(d_address) := PendingReq_x(d_address) − 1;
                {merge internal view}
        delete ⟨sdid, ∗, ∗, ∗⟩ from WView_x;
        for ⟨child, scons, wcons, gateway-set⟩ in iview do
            if ¬∃⟨child, ∗, ∗, ∗⟩ ∈ WView_x then
                insert ⟨child, scons, wcons, gateway-set⟩ in WView_x;
            else
                for ⟨gid, ts, edge-set⟩ in gateway-set do
                    if ∃⟨gid, timestamp, ∗⟩ ∈ Gateways&Edges_x(child) ∧ ts > timestamp then
                        delete ⟨gid, ∗, ∗⟩ from Gateways&Edges_x(child);
                    endif;
                    if ¬∃⟨gid, ∗, ∗⟩ ∈ Gateways&Edges_x(child) then
                        insert ⟨gid, ts, edge-set⟩ to Gateways&Edges_x(child);
                    endif
            endif
        if PendingReq_x(d_address) = 0 then            {All pending replies are received}
            search_x(d_address);
        endif
    endif
```

Figure 15: view-query protocol: State and events of a router x. (cont.)

Constants:

$\texttt{AdjLocalRouters}_x$. ($\subseteq$ $\texttt{NodeIds}$). Set of neighbor routers in x's domain.

$\texttt{AdjForeignGateways}_x$. ($\subseteq$ $\texttt{NodeIds}$). Set of neighbor routers in other domains.

$\texttt{Ancestor}_i(x)$. ($\subseteq$ $\texttt{SuperDomainIds}$). ith ancestor of x.

Variables:

$View_x$. Dynamic view of x.

$IntraDomainRT_x$. Intra-domain routing table of x. Initially contains no entries.

$Clock_x$: Integer. Clock of x.

Events:

```
Receive_x(Update, sdid, gid, ts, edge-set) from sender
    if ∃⟨gid, timestamp, *⟩ ∈ Gateways&Edges_x(sdid) ∧ ts > timestamp then
        delete ⟨gid, *, *⟩ from Gateways&Edges_x(sdid);
    endif;
    if ¬∃⟨gid, *, *⟩ ∈ Gateways&Edges_x(sdid) then
        flood_x((Update, sdid, gid, ts, edge-set));
        insert ⟨gid, ts, edge-set⟩ to Gateways&Edges_x(sdid);
        update_parent_domains_x(level(sdid) + 1);
    endif

where
  update_parent_domains_x(startinglevel)
    for level := startinglevel to number of levels in the hierarchy  do
        sdid := Ancestor_level(x);
        if x ∈ Gateways(sdid) then
            edge-set := aggregate edges of sdid:x using View_x, IntraDomainRT_x and links of x;
            timestamp = Clock_x;
            flood_x((Update, sdid, x, timestamp, edge-set));
            delete ⟨x, *, *⟩ from Gateways&Edges_x(sdid);
            insert ⟨x, timestamp, edge-set⟩ to Gateways&Edges_x(sdid);
        endif

  Do_Update_x        {Executed periodically and upon a change in IntraDomainRT_x or links of x}
    update_parent_domains_x(1)

  Link_Recovery_x(y)        {⟨x, y⟩ is a link. Executed when ⟨x, y⟩ recovers.}
    for all ⟨sdid, *, *, *⟩ in View_x do
        if ∃i : Ancestor_i(y) = Ancestor_1(sdid) then
            for all ⟨gid, timestamp, edge-set⟩ in Gateways&Edges_x(sdid) do
                Send((Update, sdid, gid, timestamp, edge-set)) to y;
        endif

  flood_x(packet)
    for all y ∈ AdjLocalRouters_x do
        Send(packet) to y;
    for all y ∈ AdjForeignGateways_x ∧ ∃i : Ancestor_i(y) = Ancestor_1(packet.sdid) do
        Send(packet) to y;
```

Figure 16: view-update protocol: State and events of a router x.