The Viewserver Hierarchy for Inter-Domain Routing: Protocols and Evaluation*

Cengiz Alaettinoğlu†
Information Sciences Institute
University of Southern California
Marina del Rey, CA 90292
cengiz@isi.edu

A. Udaya Shankar
Institute for Advanced Computer Studies
and Department of Computer Science
University of Maryland
College Park, MD 20742
shankar@cs.umd.edu

CS-TR-3151.1 UMIACS-TR-93-98.1

March 3, 1995

Abstract

We present an inter-domain routing protocol based on a new hierarchy, referred to as the viewserver hierarchy. The protocol satisfies policy and ToS constraints, adapts to dynamic topology changes including failures that partition domains, and scales well to large number of domains without losing detail (unlike the usual scaling technique of aggregating domains into superdomains). Domain-level views are maintained by special nodes called viewservers. Each viewserver maintains a view of a surrounding precinct. Viewservers are organized hierarchically. To obtain domain-level source routes, the views of one or more viewservers are merged (upto a maximum of twice the levels in the hierarchy).

We also present a model for evaluating inter-domain routing protocols, and apply this model to compare our viewserver hierarchy against the simple approach where each node maintains a domain-level view of the entire internetwork. Our results indicate that the viewserver hierarchy finds many short valid paths and reduces the amount of memory requirement by two orders of magnitude.

*To appear in IEEE JSAC Special Issue on Global Internet

* 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, or the U.S. Government.

†This research was performed while the author was with the Computer Science Department, University of Maryland, College Park, MD 20742.

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, whereas backbones, regionals, and MANs are transit networks. Most of the networks in an internetwork are stub networks. Each network consists of nodes (hosts and routers) and links. Two networks are *neighbors* when there is one or more links between nodes in the two networks (see Figure 1).

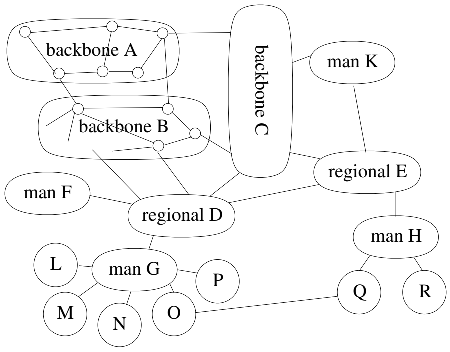

Figure 1: A portion of an internetwork.

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. Within each domain, an *intra-domain routing protocol* is executed that provides routes between source and destination nodes in the domain.

Across all domains, an *inter-domain routing protocol* is executed that provides routes between source and destination nodes in different domains. This protocol must satisfy various constraints:

(1) It must satisfy *policy constraints*, which are administrative restrictions on the inter-domain traffic [9, 13, 10, 5]. Policy constraints are of two types: *transit policies* and *source policies*. The transit policies of a domain A specify how other domains can use the resources of A (e.g. \$0.01 per packet, no traffic from domain B). The source policies of a domain A specify constraints on traffic originating from A (e.g. domains to avoid/prefer, acceptable connection

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

cost). Transit policies of a domain are public (i.e. available to other domains), whereas source policies are usually private (e.g. it may not be desirable to announce how much a domain is willing to pay for a particular service).

(2) An inter-domain routing protocol must also satisfy *type-of-service* (ToS) constraints of applications (e.g. low delay, high throughput, high reliability, minimum monetary cost). To do this, it must keep track of the types of services offered by each domain [5].

(3) Inter-domain routing protocols must 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 must also handle non-hierarchical domain interconnections at any level [10] (e.g. we do not want to hand-configure special routes as "back-doors").

(4) Inter-domain routing protocols must automatically adapt to link cost changes, node/link failures and repairs including failures that partition domains [15].

A simple straightforward approach to inter-domain routing is domain-level source routing with link-state approach [9, 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 the packets are routed to the destination using this domain-level source route and the intra-domain routing protocols of the domains crossed.

The disadvantage of this simple 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 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).

To achieve scaling, several approaches based on aggregating domains into superdomains have

[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 [9]

been proposed [19, 16, 7, 6]. These approaches have drawbacks because the aggregation results in loss of detail (discussed in Section 2).

Our protocol

In this paper, we present an inter-domain routing protocol that we proposed recently[2, 3]. It combines domain-level views with a novel hierarchical scheme. It scales well to large internetworks, and does not suffer from the problems of superdomains.

In our scheme, domain-level views are not maintained by every router but by special nodes called *viewservers*. For each viewserver, there is a subset of domains around it, referred to as the viewserver's *precinct*. The viewserver maintains the domain-level view of its precinct. This solves the scaling problem for storage requirement.

A viewserver can provide domain-level source routes between source and destination nodes in its precinct. Obtaining a domain-level source route between a source and a destination that are not in any single view, involves accumulating the views of a sequence of viewservers. To make this process efficient, viewservers are organized hierarchically in levels, and an associated addressing structure is used. Each node has a set of addresses. Each *address* is a sequence of viewserver ids of decreasing levels, starting at the top level and going towards the node. The idea is that when the views of the viewservers in an address are merged, the merged view contains domain-level routes to the node from the top level viewservers.

We handle dynamic topology changes such as node/link failures and repairs, link cost changes, and domain partitions. Gateways[5] detect domain-level topology changes affecting its domain and neighbor domains. For each domain, there is a *reporting gateway* that communicates these changes by flooding to the viewservers in a specified subset of domains; this subset is referred to as its *flood area*. Hence, the number of packets used during flooding is proportional to the size of the flood area. This solves the scaling problem for the communication requirement.

Thus our inter-domain routing protocol consists of two subprotocols: a **view-query protocol** between routers and viewservers for obtaining merged views; and a **view-update protocol** between gateways and viewservers for updating domain-level views.

[5] A node is called a gateway if it has a link to another domain.

Evaluation

Many inter-domain routing protocols have been proposed, based on various kinds of hierarchies. How do these protocols compare against each other and against the simple approach? To answer this question, 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). None of these protocols have been evaluated in a way that they can be compared against each other or the simple approach.

In this paper, we present such a model, and use it to compare our viewserver hierarchy to the simple approach. Our evaluation measures are the amount of memory required at the source and at the routers, the amount of time needed to obtain the information to construct a path, and the number of valid paths found (and their lengths) in comparison to the number of available valid paths (and their lengths) in the internetwork. We use three internetwork topologies each of size 11,110 domains (roughly the current size of the Internet). Our results indicate that the viewserver hierarchy finds many short valid paths and reduces the amount of memory requirement by two orders of magnitude.

Organization of the paper

In Section 2, we survey recent approaches to inter-domain routing. In Section 3, we present the view-query protocol for static network conditions, that is, assuming all links and nodes of the network remain operational. In Section 4, we present the view-update protocol to handle topology changes. In Section 5, we present our evaluation model and results from its application to the viewserver hierarchy. In Section 6, we conclude and describe how to add fault-tolerance and cacheing schemes to improve performance.

This paper differs from [3] in that in the latter, the view-update protocol and the evaluation model are not present, the view-query protocol is only informally described, and the evaluation results are summarized for only one internetwork topology.

2 Related Work

In this section, we survey recently proposed inter-domain routing protocols that support ToS and policy constraints for large internetworks [17, 16, 19, 11, 7, 6, 22, 1, 21, 20, 8].

Several inter-domain routing protocols (e.g. BGP [17], IDRP [16], NR [11]) are based on *path-vector* approach [18]. 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. These paths are also used to avoid routing loops. Each router requires $O(N_D \times N_D \times E_R)$ space (one entry for each of N_D domains through each of E_R neighbors; each entry contains a path whose length is bounded by N_D). For each destination, a router exchanges its best valid path[6] 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 [19]. By exchanging k paths to each destination, the probability of detecting a valid path for each source can be increased.

The most common approach to solve the scaling problem is to use *superdomains* (e.g. IDPR [19], IDRP [16], Nimrod [7, 6]). Superdomains extend the idea of *area hierarchy* [12]. Here, domains are grouped hierarchically into superdomains: "close" domains are grouped into level 1 superdomains, "close" level 1 superdomains are grouped into level 2 superdomains, and so on. A router maintains a view that contains the domains in the same level 1 superdomain, the level 1 superdomains in the same level 2 superdomain, and so on. Thus a router maintains a smaller view than it would in the absence of hierarchy. Each superdomain has its own ToS and policy constraints derived from that of the subdomains.

There are several major problems with using superdomains. One problem is that 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 techniques are to take either the union or the intersection of the constraints of the subdomains [19][7]. Both techniques have problems. For example, if the union is taken, then a subdomain A can be forced to obey constraints of other subdomains; this may eliminate a path through A which is otherwise valid. If the intersection is taken, then a subdomain A can be forced to accept traffic it would otherwise not accept. Other problems are described in [7, 6, 1]. Some of the problems can be relaxed by having

[6] A valid path is a path that satisfies the ToS and policy constraints of the domains in the path.

[7] If the union (intersection) of the constraints are taken for policies, the superdomain enforces a policy constraint if that policy constraint is enforced by some (all) of its subdomains.

overlapping superdomains, but this increases the storage requirements drastically.

Nimrod [7, 6] and IDPR [19] use the link-state approach, domain-level source routing, and superdomains (non-overlapping superdomains for Nimrod). IDRP [16] uses path-vector approach and a variation of superdomains (which are referred to as routing domain confederations).

Reference [11] 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.

In [1], we proposed a superdomain-based protocol which always finds a valid path if one exists and never admits an invalid path. It does this by maintaining both union and intersection constraints and using a view-query protocol. If the union constraints of superdomains on a path are satisfied, then the path is valid. If the intersection constraints of a superdomain are satisfied but the union constraints are not, then there may be a valid path through this superdomain. The source queries to obtain a more detailed "internal" view of such superdomains, and searches again for a valid path. Even though this protocol scales well for realistic internetwork topologies (e.g. where each superdomain is connected to at most $\log N_D$ external domains), its worst-case storage requirement can be linear in N_D.

The landmark hierarchy [21, 20] is another approach for solving the scaling problem. Here, each router is a landmark with a radius, and routers which are within a radius away from the landmark maintain a route to it. Landmarks are organized hierarchically, such that the radius of a landmark increases with its level, and the radii of top level landmarks include all routers. A thorough study of enforcing ToS and policy constraints with this hierarchy has not been done.

The landmark hierarchy may look similar to our viewserver hierarchy, but in fact it is quite the opposite. In the landmark hierarchy, nodes within the radius of a landmark maintain a route to the landmark, but the landmark may not have a route to these nodes. In the viewserver hierarchy, a viewserver maintains routes to the nodes in its precinct.

Route fragments [8] is a mechanism to glue together precomputed partial source routes to obtain a route from a source node to a destination node. A destination route fragment, called a *route suffix*, is a sequence of domain ids from a backbone to the destination domain. A source route fragment, called a *route prefix*, is the reverse of a route suffix of the source domain. There are also *route middles*, which extend from transit domains to transit domains. A source queries a name server and obtains destination route suffixes. It then chooses an appropriate route suffix for the destination and concatenates it with its own route prefix, and uses route middles if the route suffix and route prefix do not intersect. This scheme does not handle topology changes and does not address policy and ToS constraints.

3 Viewserver Hierarchy Query Protocol

In this section, we present our scheme for static network conditions, that is, all links and nodes remain operational. The dynamic case is presented in Section 4.

Conventions: Each domain has a unique id. `DomainIds` denotes the set of domain-ids. Each node has a unique id. `NodeIds` denotes the set of node-ids. For a node u, we use $domainid(u)$ to denote the domain-id of u's domain. We use $nodeid(u)$ to denote the node-id of u. For a domain A, we use $domainid(A)$ to denote the domain-id of A. $NodeNeighbors(u)$ denotes the set of node-ids of the neighbors of u. $DomainNeighbors(A)$ denotes the set of domain-ids of the domain neighbors of A.

In our protocol, a node u uses two kinds of sends. The first kind has 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 (u, v). If the link is down, we assume that the packet is dropped.

The second kind of send has the form "Send(m) to v using $dlsr$", where m and v are as above and $dlsr$ is a domain-level source route between u and v. Here, the message is sent using the intra-domain routing protocols of the domains in $dlsr$ to reach v. We assume that as long as there is a sequence of up links connecting the domains in $dlsr$, the message is delivered to v. If u and v are in the same domain, $dlsr$ equals the empty sequence $\langle\rangle$.

Views and Viewservers

Domain-level views are maintained by special nodes called *viewservers*. Each viewserver has a *precinct*, which is a set of domains around the viewserver, and a *static view*, which is a domain-level view of the precinct and outgoing edges[8]. The static view includes the ToS and policy constraints of domains in the precinct and of domain-level edges. To handle topology changes, a viewserver also maintains a dynamic view which is described in Section 4. Formally, a viewserver x maintains the following:

$Precinct_x \subseteq$ `DomainIds`. Domain-ids whose view is maintained.

$SView_x$. Static view of x.

$$= \{\langle A,\ policy\&tos(A),\ \{\langle B,\ edge\_policy\&tos(A,B)\rangle : B \in \text{ subset of } DomainNeighbors(A)\}\rangle : A \in Precinct_x\}$$

The intention of $SView_x$ is to obtain domain-level source routes between nodes in $Precinct_x$. Hence, the choice of domains to include in $Precinct_x$ and the choice of domain-level edges to include in $SView_x$ is not arbitrary. $Precinct_x$ and $SView_x$ must be connected; that is, between any two domains in $Precinct_x$, there should be a path in $SView_x$. Note that $SView_x$ can contain edges to domains outside $Precinct_x$. We say that a domain A is *in the view* of a viewserver x, if either A is in the precinct of x, or $SView_x$ has an edge from a domain in the precinct of x to A. Note that the precincts and views of different viewservers can be overlapping, identical or disjoint.

Viewserver Hierarchy

For scaling reasons, we cannot have one large view. Thus, obtaining a domain-level source route between a source and a destination which are far away, involves accumulating views of a sequence of viewservers. To keep this process efficient, we organize viewservers hierarchically. More precisely, each viewserver is assigned a hierarchy level from $0, 1, \ldots$, with 0 being the top level in the hierarchy. A parent/child relationship between viewservers is defined as follows:

1. Every level i viewserver, $i > 0$, has a parent viewserver whose level is less than i.
2. If viewserver x is a parent of viewserver y then x's view contains y's domain and y's view contains x's domain.
3. The view of a top level viewserver contains the domains of all other top level viewservers (typically, top level viewservers are placed in backbones).

[8] Not all the domain-level edges need to be included.

Note that the third constraint does not mean that all top level viewservers have the same view. In the hierarchy, a parent can have many children and a child can have many parents. We extend the range of the parent-child relationship to ordinary nodes; that is, if $Precinct_x$ contains the domain of node u, we say that u is a child of x, and x is a parent of u. We assume that there is at least one parent viewserver for each node.

For a node u, an address is defined to be a sequence $\langle x_0, x_1, \ldots, x_t \rangle$ such that x_i for $i < t$ is a viewserver-id, x_0 is a top level viewserver-id, x_t is the id of u, and x_i is a parent of x_{i+1}. A node may have many addresses since the parent-child relationship is many-to-many. If a source wants a domain-level source route to a destination, it first queries the name servers [14] to obtain a set of addresses for the destination[9]. Then, it queries viewservers to obtain an accumulated view containing both its domain and the destination's domain. Nodes can reach the viewservers in their domains using the intra-domain routing protocol of the domain. Otherwise, we assume that nodes maintain a set of fixed domain-level source routes to viewservers.

View-Query Protocol: Obtaining Domain-Level Source Routes

We now describe how a domain-level source route is obtained.

We want a sequence of viewservers whose merged views contains both the source and the destination domains. Addresses provide a way to obtain such a sequence, by first going up in the viewserver hierarchy starting from the source node and then going down in the viewserver hierarchy towards the destination node. More precisely, let $\langle s_0, \ldots, s_t \rangle$ be an address of the source, and $\langle d_0, \ldots, d_l \rangle$ be an address of the destination. Then, the sequence $\langle s_{t-1}, \ldots, s_0, d_0, \ldots, d_{l-1} \rangle$ meets our requirements[10]. In fact, going up all the way in the hierarchy to top level viewservers may not be necessary. We can stop going up at a viewserver s_i if there is a viewserver $d_j, j < l$, such that the domain of d_j is in the view of s_i (one special case is where $s_i = d_j$).

The view-query protocol uses two message types:

- (`RequestView`, *s_address*, *d_address*)

  where *s_address* and *d_address* are the addresses for the source and the destination respectively. A `RequestView` message is sent by a source to obtain an accumulated view containing

[9] Querying the name servers can be done the same way it is done currently in the Internet.

[10] This is similiar to matching route fragments[8]. However, in our case the sequence is computed in a distributed fashion (this is needed to handle topology changes).

both the source and the destination domains. When a viewserver receives a `RequestView` message, it either sends back its view or forwards this request to another viewserver.

- (`ReplyView`, *s_address*, *d_address*, *accumview*)

  where *s_address* and *d_address* are as above and *accumview* is the accumulated view. A `ReplyView` message is sent by a viewserver to the source or to another viewserver closer to the source. The *accumview* field in a `ReplyView` message equals the union of the views of the viewservers the message has visited.

We now describe the view-query protocol in more detail (please refer to Figure 2 and 3). To obtain a domain-level source route to a destination node, the source node sends a `RequestView` packet containing the source and the destination addresses to its parent in the source address.

Constants

$FixedRoutes_u(x)$, for every viewserver-id x such that x is a parent of u,

$$= \begin{cases} \{\langle\rangle\} & \text{if } domainid(u) = domainid(x) \\ \{\langle d_1, \ldots, d_n \rangle : d_i \in \texttt{DomainIds}\}.\ \text{Set of domain-level routes to } x & \text{otherwise} \end{cases}$$

Events

$RequestView_u(s\_address,\ d\_address)$ {Executed when u wants a valid domain-level source route}
  Let $s\_address$ be $\langle s_0, \ldots, s_{t-1}, s_t \rangle$, and $dlsr \in FixedRoutes_u(s_{t-1})$;
  Send(`RequestView`, *s_address*, *d_address*) to s_{t-1} using *dlsr*

$Receive_u$(`ReplyView`, *s_address*, *d_address*, *accumview*)
  Choose a valid domain-level source route using *accumview*;
  If a valid route is not found
    Execute $RequestView_u$ again with another source address and/or destination address

Figure 2: View-query protocol: Events and state of a source u.

Upon receiving a `RequestView` packet, a viewserver x checks if the destination domain is in its precinct[11]. If it is, x sends back its view in a `ReplyView` packet. If it is not, x forwards the request packet to another viewserver as follows (details in Figure 3): x checks whether the domain of any viewserver in the destination address is in its view. If there is such a domain, x sends the `RequestView` packet to the last such one in the destination address. Otherwise x is a viewserver in the source address, and it sends the packet to its parent in the source address.

When a viewserver x receives a `ReplyView` packet, it merges its view to the accumulated view in the packet. Then it sends the `ReplyView` packet towards the source node in the same way it

[11] Even though the destination can be in the view of x, its policies and ToS's are not in the view if it is not in the precinct of x.

Constants

$Precinct_x$. Precinct of x.

$SView_x$. Static view of x.

Events

$Receive_x$(`RequestView`, $s\_address$, $d\_address$)
    Let $d\_address$ be $\langle d_0, \ldots, d_t \rangle$;
    if $domainid(d_t) \notin Precinct_x$ then
        $forward_x$(`RequestView`, $s\_address$, $d\_address$, $\{\}$);
    else $forward_x$(`ReplyView`, $d\_address$, $s\_address$, $SView_x$);   {addresses are switched}
    endif

$Receive_x$(`ReplyView`, $s\_address$, $d\_address$, $view$)
    $forward_x$(`ReplyView`, $s\_address$, $d\_address$, $view \cup SView_x$)

where procedure $forward_x(type,\ s\_address,\ d\_address,\ view)$
    Let $s\_address$ be $\langle s_0, \ldots, s_t \rangle$, $d\_address$ be $\langle d_0, \ldots, d_l \rangle$;
    if $\exists i : domainid(d_i)$ in $SView_x$ then
        Let $i = \max\{j : domainid(d_j)$ in $SView_x\}$;
        $target := d_i$;
    else $target := s_i$ such that $s_{i+1} = nodeid(x)$;
    endif;
    $dlsr$:= choose a route to $domainid(target)$ from $domainid(x)$ using $SView_x$;
    if $type$ = `RequestView` then
        Send(`RequestView`, $s\_address$, $d\_address$) to $target$ using $dlsr$;
    else Send(`ReplyView`, $s\_address$, $d\_address$, $view$) to $target$ using $dlsr$;
    endif

Figure 3: View-query protocol: Events and state of a viewserver x.

would send a `RequestView` packet towards the destination node (i.e. the roles of the source address and the destination address are interchanged).

When the source receives a `ReplyView` packet, it chooses a valid path using the *accumview* in the packet. If it does not find a valid path, it can try again using a different source and/or destination address. Note that the source does not have to throw away the previous accumulated views; it can merge them all into a richer accumulated view. In fact, it is easy to change the protocol so that the source can also obtain views of individual viewservers to make the accumulated view even richer.

Above we have described one possible way of obtaining the accumulated views. There are various other possibilities, for example: (1) restricting the `ReplyView` packet to take the reverse of the path that the `RequestView` packet took; (2) having `ReplyView` packets go all the way up in the viewserver-hierarchy for a richer accumulated view; (3) having the source poll the viewservers directly instead of the viewservers forwarding request/reply messages to each other;

(4) not including non-transit stub domains other than the source and the destination domains in the *accumview*; (5) including some source policy constraints and ToS requirements in the `RequestView` packet, and having the viewservers filter out some domains.

4 Update Protocol for Dynamic Network Conditions

In this section, we first examine how topology changes such as link/node failures, repairs and cost changes, map into domain-level topology changes. Second, we describe how domain-level topology changes are detected and communicated to viewservers, i.e. the view-update protocol. Third, we modify the view-query protocol appropriately.

Mapping Topology Changes to Domain-Level Topology Changes

Costs are associated with domain-level edges. The cost of the domain-level edge (A, B) equals a vector of values if the link is up; each cost value indicates how expensive it is to cross domain A to reach domain B according to some criteria such as delay, throughput, reliability, etc. The cost equals ∞ if all links from A to B are down. Each cost value of a domain-level edge (A, B) can be derived from the cost values of the intra-domain routes in A and links from A to B [4].

Link cost changes and link/node failures and repairs correspond to cost changes, failures and repairs of domain-level edges. Link/node failures can also partition a domain into cells[15]. A *cell* is a maximal subset of nodes of a domain that can reach each other without leaving the domain. In the same way, link/node repairs may merge cells into bigger cells. We identify a cell with the minimum node-id of the gateways in the cell[12]. In this paper, for uniformity we treat an unpartitioned domain as a domain with one cell.

If a domain gets partitioned, its vertex in the domain-level views is split into as many pieces as there are cells. And when the cells merge, the corresponding vertices are merged.

Since a domain can be partitioned into many cells, domain-level source routes now include cell-ids as well. To reach the next domain cell in a domain-level source route, the intra-domain routing protocol of a domain should keep track of the domain cells reachable through each of its gateways.

[12] Our cells are like the domain components of IDPR[19].

View-Update Protocol: Updating Domain-Level Views

Viewservers do not communicate with each other to maintain their views. Gateways detect and communicate domain-level topology changes to viewservers. Each gateway periodically (and optionally after a change in the intra-domain routing table) inspects its intra-domain routing table and determines the cell to which it belongs. For each cell, only the gateway whose node-id is the cell-id (i.e. the gateway with the minimum node-id) is responsible for communicating domain-level topology changes. We refer to this gateway as the *reporting gateway*. Reporting gateways are also responsible for informing the viewservers of the creation and deletion of cells.

The communication between a reporting gateway and viewservers is done by flooding over a set of domains. This set is referred to as the *flood area*[13]. The topology of a flood area must be a connected graph.

Due to the nature of flooding, a viewserver can receive information out of order for a domain cell. In order to avoid old information replacing new information, each reporting gateway includes successively increasing time stamps in the messages it sends.

Due to node and link failures, communication between a reporting gateway and a viewserver can fail, resulting in the viewserver having out-of-date information. To eliminate such information, a viewserver deletes any information about a domain cell if it is older than a *time-to-die* period. We assume that gateways send messages more often than the time-to-die value (to avoid false removal).

When a viewserver learns of a new domain cell, it adds it to its view. To avoid adding a domain cell which was just deleted[14], when a viewserver receives a delete domain cell request, it only marks the domain cell as deleted and removes the entry after the time-to-die period.

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

- (`UpdateCell`, *domainid*, *cellid*, *timestamp*, $floodarea$, *ncostset*)
  is sent by the reporting gateway to inform the viewservers about current domain-level edge costs of its cell. Here, *domainid*, *cellid*, and *timestamp* indicate the domain, the cell and the time stamp of the reporting gateway, *ncostset* contains a cost for each domain level edge of the domain, and $floodarea$ is the set of domains that this message is to be sent over.

[13] For efficiency, the flood area can be implemented by a hop-count and some forwarding limits (e.g. do not flood beyond backbones).

[14] If the domain cell was removed, the timestamp for that domain cell is also lost.

- (`DeleteCell`, *domainid*, *cellid*, *timestamp*, *floodarea*)

  where the parameters are as in the `UpdateCell` message. It is sent by a reporting gateway when it becomes non-reporting (because its cell expanded to include a gateway with lower id).

Constants:

$LocalViewservers_g$. ($\subseteq$ `NodeIds`). Set of viewservers in g's domain.

$LocalGateways_g$. ($\subseteq$ `NodeIds`). Set of gateways in g's domain excluding g.

$AdjForeignGateways_g$. ($\subseteq$ `NodeIds`). Set of adjacent gateways in other domains.

$FloodArea_g$. ($\subseteq$ `DomainIds`). The flood area of the domain (includes domain of g).

Variables:

$IntraDomainRT_g$. Intra-domain routing table of g. Initially contains no entries.

$CellId_g$: `NodeIds`. The id of g's cell. Initially $= \infty$

$Clock_g$: Integer. Clock of g.

Figure 4: State of a gateway g.

The state maintained by a gateway g is listed in Figure 4. Note that $LocalViewservers_g$ and $LocalGateways_g$ can be empty. $IntraDomainRT_g$ contains a route (next-hop or source) for every reachable node of the domain and for every reachable neighbor domain cell. We assume that consecutive reads of $Clock_g$ return increasing values.

Constants:

$Precinct_x$. Precinct of x.

$SView_x$. Static view of x.

$TimeToDie_x$: Integer. Time-to-die value.

Variables:

$DView_x$. Dynamic view of x.

$$= \{\langle A{:}g,\ timestamp,\ expirytime,\ deleted,\ \{\langle B{:}h,\ cost\rangle : B \in DomainNeighbors(A) \ \wedge\ h \in \texttt{NodeIds} \cup \{*\}\ \}\rangle : A \in Precinct_x \ \wedge\ g \in \texttt{NodeIds}\}$$

$Clock_x$: Integer. Clock of x.

Figure 5: State of a viewserver x.

The state maintained by a viewserver x is listed in Figure 5. $DView_x$ is the dynamic part of x's view. We use *A:g* to denote the cell g of domain A. For each domain cell known to x, $DView_x$ stores a *timestamp* field which equals the largest timestamp received for this domain cell, an *expirytime* field which equals the end of the time-to-die period for this domain cell, a *deleted* field which marks whether or not the domain cell is deleted, and a cost set which indicates a cost for every domain level edge of the domain in $SView_x$. The cell-id of a neighbor domain equals $*$ if no cell of the neighbor domain is reachable.

The events of gateway g and a viewserver x are specified in Appendix A.

Changes to View-Query Protocol

We now enumerate the changes needed to adapt the view-query protocol to the dynamic case (the formal specification is omitted for space reasons).

Due to link and node failures, `RequestView` and `ReplyView` packets can get lost. Hence, the source may never receive a `ReplyView` packet after it initiates a request. Thus, the source should try again after a time-out period.

When a viewserver sends a message to a node whose domain is partitioned, it should send a copy of the message to each cell of the domain. This is because a viewserver does not know which cell contains the node.

When a viewserver receives a `RequestView` message, it should reply with its views only if the destination domain is in its precinct and its dynamic view contains a path to the destination. Similarly during forwarding of `RequestView` and `ReplyView` packets, a viewserver, when checking whether a domain is in its view, should also check if its dynamic view contains a path to it.

Note that the internetwork may partition in a way that a cell may not be in the view of any viewserver, though it is reachable by other cells. In this case, the view-query protocol will fail to discover a route to the nodes in that cell. One way to solve this problem is to dynamically change the viewserver precincts and the flood areas of domains. This is outside the scope of this paper.

5 Evaluation

Many inter-domain routing protocols have been proposed, based on various kinds of hierarchies. How do these protocols compare against each other and against the simple approach? To answer this question, 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) for inter-domain routing protocols.

In this section, we first present such a model, and then use the model to evaluate our viewserver hierarchy and compare it to the simple approach. Our evaluation measures are the amount of memory required at the source and at the routers, the amount of time needed to obtain the information to construct a path, and the number of paths found out of the total number of valid paths.

Even though the model described here can be applied to other inter-domain routing protocols, we have not done so, and hence have not compared them against our viewserver hierarchy. This is because of lack of time, and because precise definitions of the hierarchies in these protocols is not available. For example, to do a fair evaluation of IDPR[19], we need precise guidelines for how to group domains into super-domains, and how to choose between the union and intersection methods when defining policy/ToS constraints of super-domains. In fact, these protocols have not been evaluated in a way that we can compare them to the viewserver hierarchy. To the best of our knowledge, this paper is the first to evaluate a hierarchical inter-domain routing protocol against explicitly stated policy constraints.

5.1 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.; these should be connected hierarchically (e.g. regionals to backbones), but "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 is in the east coast). 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. 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 $\mathtt{r_0.r_1.\cdots.r_{i-1}}$ (where $i>0$). Containment is transitive. Thus region `1.2.3.4` is contained in regions `1.2.3`, `1.2` and `1`.

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 6, 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 6, 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 6, 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

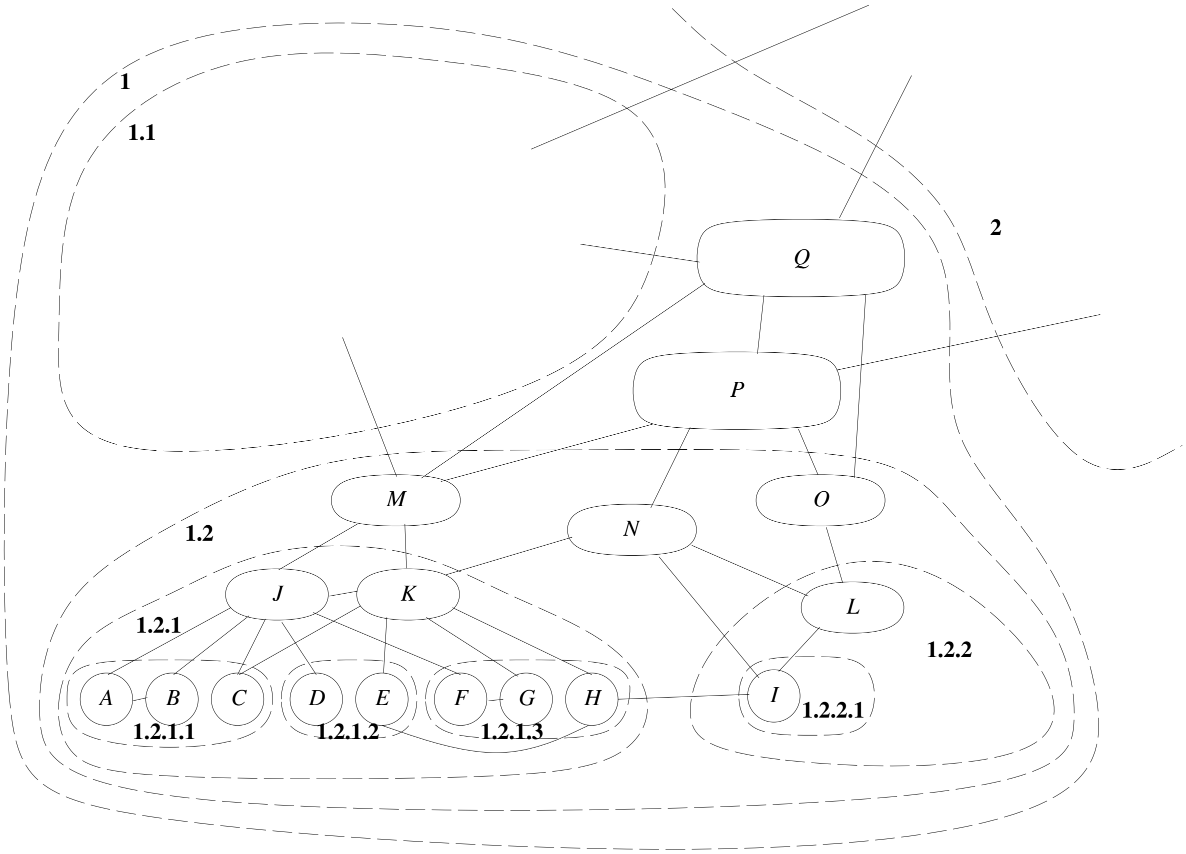

Figure 6: Regions

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.

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 are allowed to be red.

Computing Evaluation Measures

The evaluation measures of most interest for an inter-domain routing protocol are its memory and time requirements, and the number of valid paths it finds (and their lengths) in comparison to the number of available valid paths (and their lengths) in the internetwork (e.g. could it find the shortest valid path in the internetwork).

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, and (2) there exists a valid path from the source domain to the destination domain in the internetwork topology. (Note that the simple scheme would always find such a path.)

For a source-destination pair, we refer to the length of the shortest valid path in the internetwork topology as the *shortest-path length*. Since the number of paths between a source-destination pair is potentially very large (factorial in the number of domains), and we are not interested in the paths that are too long, we only count the number of paths whose lengths are not more than the shortest-path-length plus 2.

The evaluation measures described above are protocol independent. However, there are also important evaluation measures that are protocol dependent (e.g. number of levels traversed in some particular hierarchy). Because of this we postpone the precise definitions of the evaluation measures to the next subsection (their definition is dependent of viewserver hierarchy).

5.2 Application to Viewserver Protocol

We have used the above model to evaluate our viewserver protocol for several different viewserver hierarchies and query methods. We first describe the different viewserver schemes evaluated. Please refer to Figure 6 in the following discussion.

The first viewserver scheme is referred to as **base**. It has exactly one viewserver in each domain. Each viewserver is identified by its domain-id. The domains in a viewserver's precinct consist of its domain and the neighboring domains. The edges in the viewserver's view consist of the edges between the domains in the precinct, and edges outgoing from domains in the precinct to domains not in the precinct. For example, the precinct of viewserver A (i.e. the viewserver in domain A) consists of domains A, B, J; the edges in the view of viewserver A consists of domain-level edges $(A, B), (A, J), (B, J), (J, M), (J, K), (J, F), (J, D)$, and (J, C).

As for the viewserver hierarchy, a viewserver's level is defined to be the class of its domain. That is, a viewserver in a class i domain is a level i viewserver. For each level i viewserver, $i > 0$, its parent viewserver is chosen randomly from the level $i-1$ viewservers in the parent region such that there is a domain-level edge between the viewserver's domain and the parent viewserver's domain. For example, for viewserver C, we can pick viewserver J or K; suppose we pick J. For viewserver J, we have no choice but to pick M (N and O are not connected to J). For M, we pick P (out of P and Q).

We use only one address for each domain. The viewserver-address of a stub domain is concatenation of four viewserver (i.e. domain) ids. Thus, the address of A is $P.M.J.A$. Similarly, the address of H is $P.M.K.H$. To obtain a route between A and H, it suffices to obtain views of viewservers A, J, K, H.

The second viewserver scheme is referred to as **base-QT** (where the QT stands for "query upto top"). It is identical to *base* except that during the query protocol all the viewservers in the source and the destination addresses are queried. That is, to obtain a route between A and H, the views of A, J, M, P, K, H are obtained.

The third viewserver scheme is referred to as **locals**. It is identical to *base* except that now a viewserver's precinct also contains domains that have the same region as the viewserver's domain. That is, the precinct of viewserver A has the domains A, B, J, C. Note that in this scheme a viewserver's view is not necessarily connected. For example, if the edge (C, J) is removed, the view of viewserver A is no longer connected. (In Section 3, we said that the view of a viewserver should be connected. Here we have relaxed this condition to simplify testing.)

The fourth viewserver scheme is referred to as **locals-QT**. It is identical to *locals* except that during the query protocol all the viewservers in the source and the destination addresses are queried.

The fifth viewserver scheme is referred to as **vertex-extension**. It is identical to *base* except that viewserver precincts are extended as follows: Let P denote the precinct of a viewserver in the *base* scheme. For each domain X in P, if there is an edge from domain X to domain Y and Y is not in P, domain Y is added to the precinct; among Y's edges, only the ones to domains in P are added to the view. In the example, domains M, K, F, D are added to the precinct of A, but outgoing edges of these domains to other domains are not included (e.g. (F, G) is not included). The advantage of this scheme is that even though it increases the precinct size by a factor of E_D (where E_D is the average number of neighbor domains to a domain), it increases the number of edges stored in the view by a factor less than 2. (In fact, if the same edge cost and edge policies are used for both directions of domain-level edges, then the only other information that needs to be stored by the viewservers is the policy constraints of the newly added domains.)

The sixth viewserver scheme is referred to as **full-QT**. It is constructed in the same way as *vertex-extension* except that the *locals* scheme is used instead of *base* scheme to define the P in the construction. In **full-QT**, during the query protocol all the viewservers in the source and the destination addresses are queried.

We also looked at two other schemes, **vertex-extension-QT** and **full**, for which the results were very close to the *vertex-extension* and the *full-QT* schemes respectively. Hence, we do not present any results for these two schemes.

In all the above viewserver schemes, we have used the same hierarchy for both domain classes and viewservers. In practice, not all domains need to have a viewserver, and a viewserver hierarchy different from the domain class hierarchy can be deployed. However, there is an advantage of having a viewserver in each domain; that is, source nodes do not require fixed domain-level source routes to their parent viewservers (in the view-query protocol). This reduces the amount of hand configuration required. In fact, the *base* scheme does not require any hand configuration, viewservers can decide their precincts from the intra-domain routing tables, and nodes can use intra-domain routes to reach parent viewservers.

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[15]	% 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 1000 source-destination pairs. For brevity, we use *spl* to refer to the *shortest-path length* (i.e. the length of the shortest valid path in the internetwork topology). The minimum *spl* of these pairs was 2, the maximum *spl* was 13, and the average *spl* was 6.8. Table 2 lists for each viewserver scheme (1) the minimum, average and maximum precinct sizes, (2) the minimum, average and maximum merged view sizes, and (3) the minimum, average and maximum number of viewservers queried.

The precinct size indicates the memory requirement at a viewserver. More precisely, the memory requirement at a viewserver is $O(\text{precinct size} \times E_D)$, except for the *vertex-extension* and *full-QT* schemes. In these schemes, the memory requirement is increased by a factor less than two. Hence the *vertex-extension* scheme has the same order of viewserver memory requirement as the *base* scheme and the *full-QT* scheme has the same order of viewserver memory requirement as the *locals*

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

Scheme	Precinct Size	Merged View Size	No. of Viewservers Queried
base	2 / 3.2 / 68	7 / 71.03 / 101	3 / 7.51 / 8
base-QT	2 / 3.2 / 68	30 / 76.01 / 101	8 / 8.00 / 8
locals	2 / 52.0 / 103	3 / 95.40 / 143	2 / 7.42 / 8
locals-QT	2 / 52.0 / 103	43 / 101.86 / 143	8 / 8.00 / 8
vertex-extension	3 / 19.2 / 796	23 / 362.15 / 486	3 / 7.51 / 8
full-QT	11 / 102.9 / 796	228 / 396.80 / 519	8 / 8.00 / 8

Table 2: Precinct sizes, merged view sizes, and number of viewservers queried for Internetwork 1.

scheme.

The merged view size indicates the memory requirement at a source; i.e. the memory requirement at a source is $O(\text{merged view size} \times E_D)$ except for the *vertex-extension* and *full-QT* schemes. Note that the source does not need to store information about red and non-transit domains. The numbers in Table 2 take advantage of this.

The number of viewservers queried indicates the communication time required to obtain the merged view at the source. Because the average *spl* is 6.8, the "real-time" communication time required to obtain the merged view at a source is slightly more than one round-trip time between the source and the destination.

As is apparent from Table 2, using a *QT* scheme increases the merged view size and the number of viewservers queried only by about 5%. Using a *locals* scheme increases the merged view size by about 30%. Using the *vertex-extension* scheme increases the merged view size by 5 times (note that the amount of actual memory needed increases only by a factor less than 2).

The number of viewservers queried in the *locals* scheme is less than the number of viewservers queried in the *base* scheme. This is because the viewservers in the *locals* scheme have bigger precincts, and a path from the source to the destination can be found using fewer views.

Table 3 shows the average number of *spl*, *spl* + 1, *spl* + 2 length paths found for a source-destination pair by the simple approach and by the viewserver schemes. All the viewserver schemes are very close to the simple approach. The *vertex-extension* and *full-QT* schemes are especially close (they found 98% of all paths). Table 3 also shows the number of pairs for which the viewserver schemes did not find a path (ranging from 1.4% to 5.9% of the source-destination pairs), and the number of pairs for which the viewserver schemes found longer paths. For these pairs, more viewserver addresses need to be tried. Note that the *locals* and *vertex-extension* schemes decrease the number of these pairs substantially (adding *QT* yields further improvement). Our policy constraints are source and destination domain independent. Hence, even a class 2 domain, if it is red, can not carry traffic to a class 3 domain to which it is connected. We believe that these figures would improve with policies that are dependent on source and destination domains.

We examined the shortest valid paths between the source-destination pairs for which the viewserver schemes failed to find paths. We found out that all these paths were very long (11 domain hops or more for *full-QT*) and very non-hierarchical (i.e. contained many links between class 3 domains).

	Number of paths found			No. of pairs	No. of pairs
Scheme	spl	$spl+1$	$spl+2$	with no path	with longer paths
simple	2.51	18.48	131.01	N/A	N/A
base	2.41	15.84	99.42	59	3 by 1.33 hops
base-QT	2.41	15.86	100.16	54	3 by 1.33 hops
locals	2.41	16.17	103.54	29	3 by 1 hop
locals-QT	2.41	16.29	105.02	20	3 by 1 hop
vertex-extension	2.51	18.38	128.19	22	0 by 0 hops
full-QT	2.50	18.40	128.90	14	0 by 0 hops

Table 3: Number of paths found for Internetwork 1.

As is apparent from Table 3 and Table 2, the *locals* scheme does not find many more extra paths than the *base* scheme even though it has larger precinct and merged view sizes. Hence it is not recommended. The *vertex-extension* scheme is the best, but even *base* is adequate since it finds many paths.

We have repeated the above evaluations for two other internetworks and obtained similar conclusions. The results are in Appendix B.

6 Concluding Remarks

We presented a hierarchical inter-domain routing protocol that 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 uses partial domain-level views to achieve scaling in space requirement. It floods domain-level topological changes over limited flood areas to achieve scaling in communication requirement.

It does not abstract domains into superdomains; hence it does not lose any domain-level detail in ToS and policy information. It merges a sequence of partial views to obtain domain-level source routes between nodes which are far away. The number of views that need to be merged is bounded by twice the number of levels in the hierarchy.

Another advantage of our protocol is that it does not tightly bind addresses of nodes to their locations in the internetwork. Rather, addresses are bound to indirect providers of information needed for route computation.

To evaluate and compare inter-domain routing protocols against each other and against the simple approach, we presented a model in which one can define internetwork topologies, policy/ToS constraints, inter-domain routing hierarchies, and evaluation measures. We applied this model to evaluate our viewserver hierarchy and compared it to the simple approach. Our results indicate that the viewserver hierarchy finds many short valid paths and reduces the amount of memory requirement by two orders of magnitude.

Our protocol recovers from fail-stop failures of viewservers and gateways. When a viewserver fails, an address which includes the viewserver's id becomes useless. This deficiency can be overcome by replicating each viewserver at different nodes of the domain; in this case a viewserver fails only if all nodes implementing it fail.

One drawback of our protocol is that to obtain a domain-level 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 [9, 19, 7, 6, 11]. There are several ways to reduce this setup overhead. First, domain-level source routes to frequently used destinations can be cached. Second, views of frequently queried viewservers can be replicated at "mirror" viewservers close to the source domain. Third, connection setup also involves traversing the name server hierarchy (to obtain destination addresses from names). By integrating the name server hierarchy with the viewserver hierarchy, we may be able to do both operations simultaneously.

The viewserver hierarchy takes advantage of the structure found in realistic internetwork topologies. It would be interesting to investigate applications of the viewserver hierarchy to arbitrary internetworks where the interconnection of networks may not be as hierarchical, but rather is more like a mesh. We are also studying techniques to apply when a valid path cannot be found using one pair of addresses. In particular, we are investigating the use of multiple addresses and heuristics to query viewservers that are not on any address but whose views may help find valid paths.

There were several drawbacks of our evaluation. We only considered simple binary policy/ToS constraints as opposed to more general policy/ToS constraints such as delay. We only evaluated the viewserver schemes using the hierarchical internetwork topologies that mimicked the Internet as opposed to more general topologies that included more mesh-like interconnections. We did not evaluate the communication capacity requirements of our protocols. We believe the communication capacity requirements of our protocols will be much less than the simple approach where the topology changes are flooded to all the routers in the internetwork.

References

[1] C. Alaettinoğlu and A. U. Shankar. Hierarchical inter-domain routing protocol with on-demand ToS and policy resolution. In *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 *IEEE INFOCOM '94*, Toronto, Canada, June 1994.

[4] A. Bar-Noy and M. Gopal. Topology distribution cost vs. efficient routing in large networks. In *ACM SIGCOMM '90*, pages 242–252, Philadelphia, Pennsylvania, September 1990.

[5] L. Breslau and D. Estrin. Design of inter–administrative domain routing protocols. In *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] J. N. Chiappa. A new ip routing and addressing architecture. Internet Draft., 1992. Available by anonymous ftp from `research.ftp.com:pub/nimrod`.

[8] D. Clark. Route fragments, a routing proposal. Big-Internet mailing list., July 1992. Available by anonymous ftp from `munnari.oz.au:big-internet/list-archive`.

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

[10] D. Estrin. Policy requirements for inter administrative domain routing. Request for Comment RFC-1125, Network Information Center, November 1989.

[11] D. Estrin, Y. Rekhter, and S. Hotz. Scalable inter-domain routing architecture. In *ACM SIGCOMM '92*, pages 40–52, Baltimore, Maryland, August 1992.

[12] F. Kamoun and L. Kleinrock. Stochastic performance evaluation of hierarchical routing for large networks. *Computer Networks and ISDN Systems*, 1979.

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

[14] P. V. Mockapetris. Domain names - concepts and facilities. Request for Comment RFC-1034, Network Information Center, November 1987.

[15] R. Perlman. Hierarchical networks and subnetwork partition problem. *Computer Networks and ISDN Systems*, 9:297–303, 1985.

[16] Y. Rekhter. Inter-domain routing protocol (idrp). *Journal of Internetworking Research and Experience*, 4:61–80, 1993.

[17] Y. Rekhter and T. Li. A border gateway protocol 4 (bgp-4). Request for Comment RFC-1654, Network Information Center, July 1994.

[18] K. G. Shin and M. Chen. Performance analysis of distributed routing strategies free of ping-pong-type looping. *IEEE Transactions on Computers*, 1987.

[19] M. Steenstrup. An architecture for inter-domain policy routing. Request for Comment RFC-1478, Network Information Center, July 1993.

[20] 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.

[21] P. F. Tsuchiya. The landmark hierarchy:a new hierarchy for routing in very large networks. In *ACM SIGCOMM '88*, August 1988.

[22] P. F. Tsuchiya. Efficient and robust policy routing using multiple hierarchical addresses. In *ACM SIGCOMM '91*, pages 53–65, Zurich, Switzerland, September 1991.

A View-Update Protocol Event Specifications

The events of gateway g are specified in Figure 7. When a gateway g recovers, $CellId_g$ is set to *nodeid*(g). Thus, when g next executes $Update_g$, it sends either an `UpdateCell` or a `DeleteCell` message to viewservers, depending on whether it is no longer the minimum id gateway in its cell. Sending a `DeleteCell` message is essential. Because prior to the failure, g may have been the smallest id gateway in its cell. Hence, some viewserver's may still contain an entry for its old domain cell.

The events of a viewserver x are specified in Figure 5. When a viewserver x recovers, $DView_x$ is set to $\{\}$. Its view becomes up-to-date as it receives new information from reporting gateways (and remove false information with the time-to-die period).

$Update_g$ {Executed periodically and also optionally upon a change in $IntraDomainRT_g$}
{Determines the id of g's cell and initiates `UpdateCell` and `DeleteCell` messages if needed.}
$OldCellId = CellId_g$;
$CellId_g :=$ compute cell id using $LocalGateways_g$ and $IntraDomainRT_g$;
if $nodeid(g) = CellId_g$ then
$ncostset :=$ compute costs for each neighbor domain cell using $IntraDomainRT_g$;
$flood_g$((`UpdateCell`, $domainid(g)$, $CellId_g$, $Clock_g$, $FloodArea_g, ncostset$));
endif
if $nodeid(g) = OldCellId \neq CellId_g$ then
$flood_g$((`DeleteCell`, $domainid(g)$, $nodeid(g)$, $Clock_g$, $FloodArea_g$));
endif

$Receive_g(packet)$ {either an `UpdateCell` or a `DeleteCell` packet}
$flood_g(packet)$

where procedure $flood_g(packet)$
if $domainid(g) \in packet.floodarea$ then
{remove domain of g from the flood area to avoid infinite exchange of the same message.}
$packet.floodarea := packet.floodarea - \{domainid(g)\}$;
for all $h \in LocalGateways_g \cup LocalViewservers_g$ do
Send($packet$) to h using $\langle\rangle$;
endif
for all $h \in AdjForeignGateways_g \wedge domainid(h) \in packet.floodarea$ do
Send($packet$) to h;

Gateway Failure Model: A gateway can undergo failures and recoveries at anytime. We assume failures are fail-stop (i.e. a failed gateway does not send erroneous messages). When a gateway g recovers, $CellId_g$ is set to $nodeid(g)$.

Figure 7: View-update protocol: Events of a gateway g.

B 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 (a different seed is used for the random number generation).

Our evaluation measures were computed for a set of 1000 source-destination pairs. The minimum *spl* of these pairs was 2, the maximum *spl* was 13, and the average *spl* was 7.2.

Table 4 and Table 5 shows the results. Similar conclusions to Internetwork 1 hold for Internetwork 2. In Table 5, the reason that *local* and QT schemes have more pairs with longer paths than the *base* scheme is that these schemes found some paths (which are not shortest) for some pairs for which the *base* scheme did not find any path.

```
Receive_x(UpdateCell, did, cid, ts, FloodArea, ncset)
  if did ∈ Precinct_x then
    if ∃⟨did:cid, timestamp, expirytime, deleted, ncostset⟩ ∈ DView_x ∧
      ts > timestamp then          {received is more recent; delete the old one}
        delete ⟨did:cid, timestamp, expirytime, deleted, ncostset⟩ from DView_x;
    endif
    if ¬∃⟨did:cid, timestamp, expirytime, deleted, ncostset⟩ ∈ DView_x then
        Choose ncostset from ncset using SView_x;
        insert ⟨did:cid, ts, Clock_x + TimeToDie_x, false, ncostset⟩ to DView_x;
    endif
  endif
Receive_x(DeleteCell, did, cid, ts, floodarea)
  if did ∈ Precinct_x then
    if ∃⟨did:cid, timestamp, expirytime, deleted, ncostset⟩ ∈ DView_x ∧
      ts > timestamp then          {received is more recent; delete the old one}
        delete ⟨did:cid, timestamp, expirytime, deleted, ncostset⟩ from DView_x;
    endif
    if ¬∃⟨did:cid, timestamp, expirytime, deleted, ncostset⟩ ∈ DView_x then
        insert ⟨did:cid, ts, Clock_x + TimeToDie_x, true, {}⟩ to DView_x;
    endif
  endif
Delete_x        {Executed periodically to delete entries older than the time-to-die period}
  for all ⟨A:g, tstamp, expirytime, deleted, ncset⟩ ∈ DView_x ∧ expirytime < Clock_x do
    delete ⟨A:g, tstamp, expirytime, deleted, ncset⟩ from DView_x;
```

Viewserver Failure Model: A viewserver can undergo failures and recoveries at anytime. We assume failures are fail-stop. When a viewserver x recovers, $DView_x$ is set to $\{\}$.

Figure 8: View update events of a viewserver x.

Scheme	Precinct Size	Merged View Size	No. of Viewservers Queried
base	2 / 3.2 / 76	4 / 66.62 / 96	3 / 7.55 / 8
base-QT	2 / 3.2 / 76	29 / 72.76 / 96	8 / 8.00 / 8
locals	3 / 69.8 / 149	4 / 101.32 / 148	2 / 7.36 / 8
locals-QT	3 / 69.8 / 149	35 / 110.32 / 152	8 / 8.00 / 8
vertex-extension	3 / 19.47 / 817	15 / 339.60 / 469	3 / 7.55 / 8
full-QT	11 / 135.2 / 817	186 / 402.51 / 503	8 / 8.00 / 8

Table 4: Precinct sizes, merged view sizes, and no of viewservers queried for Internetwork 2.

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

Scheme	Number of paths found			No. of pairs with no path	No. of pairs with longer paths
	spl	$spl+1$	$spl+2$		
simple	2.21	13.22	74.30	N/A	N/A
base	1.98	8.20	34.40	123	13 by 1.08 hops
base-QT	1.98	8.36	35.62	110	15 by 1.13 hops
locals	2.08	9.18	40.50	97	23 by 1.39 hops
locals-QT	2.08	9.38	42.08	67	23 by 1.30 hops
vertex-extension	2.18	12.57	64.98	19	6 by 1 hop
full-QT	2.19	12.85	67.37	4	4 by 1 hop

Table 5: Number of paths found for Internetwork 2.

class 3 domains are red. Hence, we expect more valid paths between source and destination pairs.

Our evaluation measures were computed for a set of 1000 source-destination pairs. The minimum spl of these pairs was 2, the maximum spl was 10, and the average spl was 5.93.

Class i	No. of Domains	No. of Regions[16]	% 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.

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

Table 7 and Table 8 shows the results. Similar conclusions to Internetwork 1 and 2 hold for Internetwork 3.

Scheme	Precinct Size	Merged View Size	No. of Viewservers Queried
base	2 / 3.5 / 171	5 / 134.41 / 206	3 / 7.26 / 8
base-QT	2 / 3.5 / 171	55 / 154.51 / 206	8 / 8.00 / 8
locals	3 / 70.17 / 171	4 / 164.16 / 257	2 / 7.09 / 8
locals-QT	3 / 70.17 / 171	57 / 191.06 / 258	8 / 8.00 / 8
vertex-extension	5 / 34.17 / 1986	18 / 601.56 / 695	3 / 7.26 / 8
full-QT	14 / 155.5 / 1986	503 / 655.79 / 743	8 / 8.00 / 8

Table 7: Precinct sizes, merged view sizes, and no of viewservers queried for Internetwork 3.

	Number of paths found			No. of pairs	No. of pairs
Scheme	*spl*	*spl* + 1	*spl* + 2	with no path	with longer paths
simple	3.34	37.55	368.97	N/A	N/A
base	2.83	24.25	178.08	17	11 by 1.09 hops
base-QT	2.87	25.53	193.41	12	8 by 1.12 hops
locals	2.87	25.62	196.33	21	8 by 1 hop
locals-QT	2.97	27.59	219.63	2	6 by 1 hop
vertex-extension	3.32	35.73	332.54	5	1 by 1 hop
full-QT	3.33	36.47	346.44	0	0 by 0 hops

Table 8: Number of paths found for Internetwork 3.

Figure 9 through Figure 11 show the number of *spl*, *spl* + 1 and *spl* + 2 length paths found by the schemes as a function of *spl* (we only show results for *spl* values for which more than 10 pairs exist). We do not include *base-QT*, *locals* and *locals-QT* schemes since they are very close to *base* scheme. As expected, as *spl* increases, the number of paths for a source-destination pair increases, and the gap between the *simple* scheme and the viewserver schemes increases.

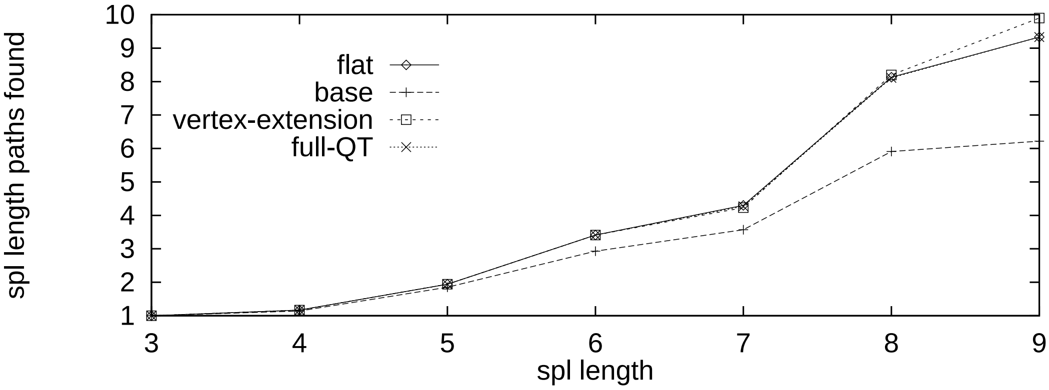

Figure 9: Number of *spl* length paths found for Internetwork 3.

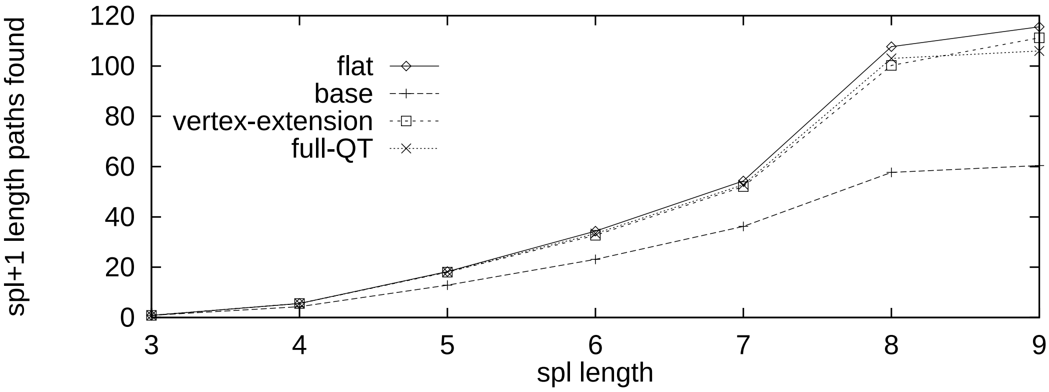

Figure 10: Number of $spl + 1$ length paths found for Internetwork 3.

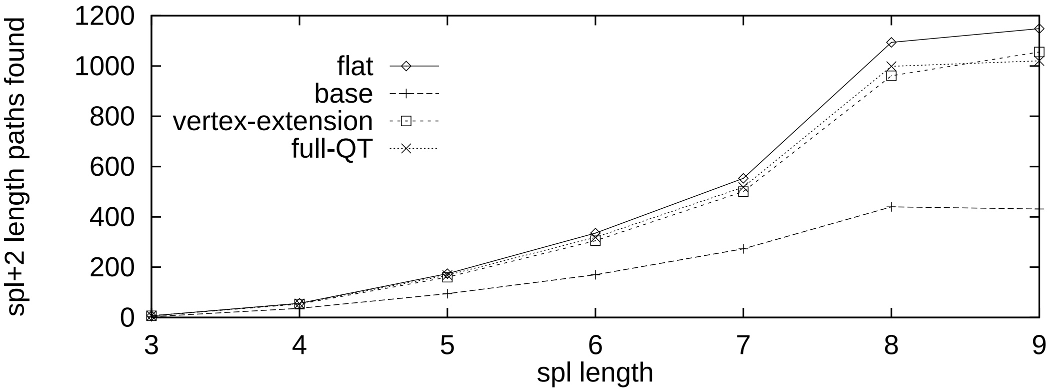

Figure 11: Number of $spl + 2$ length paths found for Internetwork 3.