The Dissertation Committee for Chen Qian
certifies that this is the approved version of the following dissertation:

A Scalable, Resilient, and Self-managing Layer-2 Network

Committee:

Simon S. Lam, Supervisor

Kenneth L. Calvert

Mohamed G. Gouda

Aloysius K. Mok

Lili Qiu

A Scalable, Resilient, and Self-managing Layer-2 Network

by

Chen Qian, B.S.; M.Phil.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2013

To Yiyi and our daughter Zhuoying

Acknowledgments

Being a student of Professor Simon Lam is definitely one of the luckiest experiences in my life. He teaches and encourages me to conduct solid research with real impact in the area of computer networking. Besides that, he has given me numerous precious advices on my career path and life. He has set a shining example of a scholar to me. In Eastern culture, a best advisor is considered another father of the student. Prof. Lam is just such a perfect advisor for me.

I would like to thank Professor Lionel Ni who has given me a tremendous amount of help when I started to do research. Without his support and guidance for my M.Phil. study, I might not be able to get admitted by the Ph.D. program of UTCS.

Professor Yunhao Liu is another great collaborator and mentor of me. He revised the very first research paper that I wrote as the first author. Since then, he has been giving me valuable advice on how to succeed in an academic career.

I would like to thank Prof. Mohamed Gouda, Prof. Al Mok, Prof. Ken Calvert, and Prof. Lili Qiu for serving my dissertation committee and their suggestions and comments throughout this dissertation.

I am fortunate to work with many great collaborators in various projects, including Jian Ma, Yanmin Zhu, Yunhuai Liu, Raymond Ngan, Quanbin Chen, Dian Zhang, Yiyang Zhao, Qian Zhang, Xi Zhou, Vinod Venkataraman, Yuanqing Zheng, Mo Li, Song Han, and Tianji Li. I thank the previous and current group members Hongkun Yang and Jaeyoun Kim for their inspiring discussion and thoughtful comments. I am happy to have many friends in and out of UTCS during these five years.

I would like to thank Xu Chen, Seungjoon Lee, and Jacobus Van der Merwe for giving me the opportunity to work as an intern at AT&T Labs. It was great fun to discuss with them about research in cloud computing and data center networks.

My parents and parents-in-law are all outstanding teachers. Although they do not work in computer science, their dedication to education and knowledge has encouraged me since I was very young. I am also indebted to them for their love, understanding, and help when it was most needed. I would thank my little one Zhuoying for bringing great joy to my life. Lastly, I thank Yiyi for sharing her life with me and her endless love, patience, and encouragement. Without her support, I would not be able to make this dissertation possible.

CHEN QIAN

The University of Texas at Austin
August 2013

A Scalable, Resilient, and Self-managing Layer-2 Network

Publication No. ____________________

Chen Qian, Ph.D.
The University of Texas at Austin, 2013

Supervisor: Simon S. Lam

Large-scale layer-2 Ethernet networks are needed for important future and current applications and services including: metro Ethernet, wide area Ethernet, data center networks, cyber-physical systems, and large data processing. However Ethernet bridging was designed for small local area networks and suffers scalability and resiliency problems for large networks. I will present the architecture and protocols of ROME, a layer-2 network designed to be backwards compatible with Ethernet and scalable to tens of thousands of switches and millions of end hosts. We first design a scalable greedy routing protocol, Multi-hop Delaunay Triangulation (MDT) routing, for delivery guarantee on any connectivity graph with arbitrary node coordinates. To achieve near-optimal routing path for greedy routing, we then present the first layer-2 virtual positioning protocol, Virtual Position on Delaunay (VPoD). We then design a stateless multicast protocol, to support group communication such as VLAN while improving switch memory scalability. To achieve efficient host discovery, we present a novel distributed hash table, Delaunay DHT (D^2HT). ROME also includes routing and host discovery protocols for a hierarchical network. ROME protocols completely eliminate broadcast. Extensive experimental results show that ROME protocols are efficient and scalable to metropolitan size. Furthermore, ROME protocols are highly resilient to network dynamics. The routing latency of ROME is only slightly higher than shortest-path latency.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Ethernet, a fundamental communication technology in data link layer (layer 2), was designed for a shared communication channel by a small group of end hosts. However, large layer-2 networks, *each* scalable to tens of thousands of switches and connecting millions of end hosts, are needed for important future and current applications and services [42] including: data center networks [20], metro Ethernet [1, 5, 21, 22], wide area Ethernet [6], cyber-physical systems [33], as well as enterprise and provider networks. As an example, the globally-distributed database for a large corporation may scale up to millions of machines across hundreds of data centers [11]. We refer to such large-scale layer-2 networks collectively as *metropolitan-scale Ethernet*.

Ethernet offers plug-and-play functionality and a flat MAC address space. Ethernet MAC addresses, being permanent and location independent, support host mobility and facilitate management functions, such as trouble shooting and access control. For these reasons, Ethernet is easy to manage. However Ethernet bridging is not scalable to a large network because it uses a spanning tree routing protocol that is highly inefficient and not resilient to failures. Switch memory scalability is another important concern because fast memory such as SRAM is power-intensive and expensive. Also, Ethernet relies on network-wide flooding for host discovery, which costs extremely high bandwidth for large networks.

Today's metropolitan and wide area Ethernet services provided by network operators are based upon a network of IP (layer-3) and MPLS routers which interconnect relatively small Ethernet LANs [21]. Adding the IP layer to perform end-to-end routing in these networks nullifies Ethernet's desirable properties. IP routing protocols (such as, RIP, OSPF, and IS-IS) are also not scalable, even though they provide shortest paths and are much more efficient than Ethernet's spanning tree protocol. More importantly, large IP networks require massive efforts by human operators to configure and manage, such as creating subnets and assigning IP addresses.

Therefore, it is desirable to have a *scalable and resilient layer-2 network that is backwards compatible with Ethernet*, i.e., its switches interact with hosts by Ethernet frames using conventional Ethernet format and semantics.

Towards this goal, Myers et al. [42] proposed replacing Ethernet broadcast for host discovery by a layer-2 distributed directory service. In 2007, replacing Ethernet broadcast by a distributed hash table (DHT) was proposed independently by Kim and Rexford [26] and Ray et al. [49]. In 2008, Kim et al. presented SEATTLE [25] which uses link-state routing, a one-hop DHT (based on link-state routing) for host discovery, and multicast trees for broadcasting to VLANs. Scalability of SEATTLE to metropolitan scale is limited by link-state broadcast. In 2009, AIR [51] was proposed to replace link state routing in SEATTLE. However, its latency was found to be larger than the latency of SEATTLE by 1.5 orders of magnitude. In 2011, VIRO [23] was proposed to replace link-state routing. To construct a rooted virtual binary tree for routing, it uses a centralized algorithm for large networks and a distributed algorithm for small networks. In all three papers [25, 51, 23], simulation experiments were performed for networks of several hundred switches.

To achieve the goal of metropolitan-scale Ethernet, we present the ROME (Routing On Metropolitan-scale Ethernet) architecture and protocols. The series of work for the ROME architecture are summarized as follows.

1. We first present a novel geographic routing protocol, Multi-hop Delaunay Triangula-

tion (MDT) routing [29], that provides guaranteed delivery for an arbitrarily connected network of nodes in a d-dimensional space, for $d \geq 2$. (Only Euclidean spaces are considered in this dissertation.) The guaranteed delivery property is proved for node locations specified by arbitrary coordinates; thus the property also holds for locations of nodes specified by inaccurate coordinates or accurate coordinates. We show experimentally that MDT routing provides a routing (distance) stretch close to 1 for nodes in 2D and 3D when coordinates specifying node locations are accurate.[1] When coordinates specifying node locations are highly inaccurate, we show that MDT routing provides a low routing (distance) stretch relative to other geographic routing protocols. Nodes may also be arbitrarily located in a virtual space with packets routed by MDT using the coordinates of nodes in the virtual space (instead of their coordinates in physical space). In this case, MDT routing still provides guaranteed delivery.

2. We present Greedy Distance Vector (GDV) [46], the extension of MDT on *virtual coordinates*. GDV is the first greedy routing protocol designed with the objective of providing near-optimal paths for any additive routing metric. To apply GDV, each node assigns itself a virtual position (position in a virtual space) such that the Euclidean distance between any pair of nodes in the virtual space is a good estimate the routing cost between them. Like DV routing, GDV selects as the next hop to destination t, a neighbor v that minimizes $c(u,x) + \tilde{D}(x,t)$ for $x \in P_u$, where $\tilde{D}(x,t)$ is the estimated routing cost from x to t from *locally computing* the distance between the virtual positions of x and t. Since $c(u,v) + \tilde{D}(v,t) \approx c(u,v) + D(v,t)$, the quality of GDV paths is expected to be close to that of optimal DV paths. We also present the first virtual positioning protocol for layer-2 networks, named Virtual Position by Delaunay (VPoD). VPoD makes use of the multi-hop DT structure, for the distributed computation of node coordinates that provide near-optimal routing path for GDV.

3. Based on MDT and GDV, we design the unicast and multicast routing protocols in

[1]Routing and distance stretch are defined later.

ROME. Unicast packet delivery in ROME is provided by GDV routing in the multi-hop DT maintained by switches. In a correct multi-hop DT, *ROME unicast routing provides guaranteed delivery of any packet to the switch that is closest to the packet's destination location* [29, 46]. We also present a *stateless multicast* protocol for group-wide broadcast in ROME. A group message is delivered using the locations of its receivers without construction of any multicast tree. Switches do not store any state for delivering group messages.

4. To route a packet from its source host to its destination host, switches need to know the MAC address of the destination host as well as its location. Such address and location resolution are together referred to as *host discovery*. We present the ROME protocols for host and service discovery using a new method, called Delaunay Distributed Hash Table (D^2HT). Unlike host discovery in conventional Ethernet, D^2HT does not use broadcast.

5. We present the ROME protocols for routing and host discovery in a hierarchical network that scales up to tens of thousands of switches and millions of hosts.

ROME is evaluated and compared using a packet-level event-driven simulator in which ROME protocols (including GDV, MDT, and VPoD) are implemented in detail. Every protocol message is routed and processed by switches hop by hop from source to destination. Experimental results show that ROME protocols are efficient and scalable. The routing latency of ROME is only slightly higher than the shortest-path latency. ROME protocols are highly resilient to network dynamics and switches quickly recover after a period of churn. To demonstrate scalability, we provide simulation performance results for ROME networks with up to 25,000 switches and 1.25 million hosts.

Chapter 2

Multi-hop Delaunay Triangulation Routing

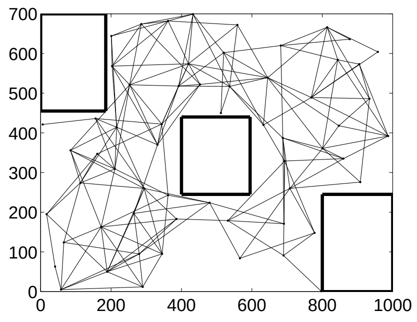
Geographic routing (also known as location-based or geometric routing) uses greedy forwarding on the set of nodes (e.g., routers, switches, or wireless sensors) with coordinates in a Euclidean space. At each step the sender sends the data message to a physical neighbor that is closest to the destination in the Euclidean space. Geographic routing provides node memory scalability because the routing state needed for greedy forwarding at each node is independent of network size. In this dissertation, we propose to use geographic routing as a scalable and resilient routing solution for layer-2 networks. However, to apply geographic routing to any connected layer-2 network, a number of problems need to be solved.

Consider a network represented by a connected graph of nodes and physical links (to be referred to as the *connectivity graph*). Greedy forwarding of a packet may be stuck at a *local minimum*, i.e., the packet is at a node closer to the packet's destination than any of the node's directly-connected neighbors. Geographic routing protocols differ mainly in their recovery methods designed to move packets out of local minima. Nodes of a layer-2 network may have physical locations in 3D [45, 4, 16, 17], or are assigned virtual coordinates in d-dimensional spaces ($d > 2$) [43, 13, 46]. For general connectivity graphs in 3D or a higher dimensional space, face routing methods designed for 2D [8, 24, 27] are not applicable. Furthermore, Durocher et al. [16] proved that there is no "local" routing protocol that provides guaranteed delivery, even under the strong assumptions of a "unit ball graph" and accurate location information. Thus, designing a geographic routing protocol that provides guaranteed delivery in d-dimensional spaces ($d > 2$) is a challenging problem.

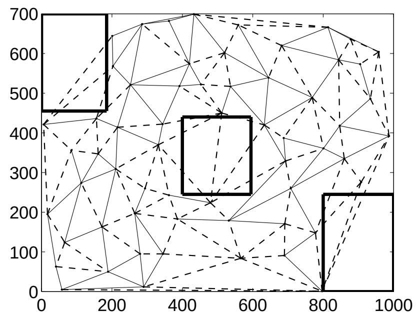
We present in this chapter a novel geographic routing protocol, MDT, that provides guaranteed delivery for a network of nodes in a d-dimensional space, for $d \geq 2$. (Only Euclidean spaces are considered in this dissertation.) The guaranteed delivery property is proved for node locations specified by arbitrary coordinates; thus the property also holds for node locations specified by inaccurate coordinates or accurate coordinates. We show experimentally that MDT routing provides a routing (distance) stretch close to 1 for nodes in 2D and 3D when coordinates specifying node locations are accurate.[1] When coordinates specifying node locations are highly inaccurate, we show that MDT routing provides a low routing (distance) stretch relative to other geographic routing protocols. Nodes may also be arbitrarily located in a virtual space with packets routed by MDT using the coordinates of nodes in the virtual space (instead of their geographic locations in physical space). In this case, MDT routing still provides guaranteed delivery.

Geographic routing in a virtual space is useful for networks without location information or networks in which the routing cost between two nodes is not proportional to the physical distance between them. We will show that geographic routing can achieve near-optimal routing paths by assigning nodes to locations in the virtual space such that the Euclidean distance between each pair of nodes in the virtual space is a good estimate of the routing cost between them. The problem is solved in Chapter 3 where we show how to (i) make use of virtual coordinates to embed routing costs in virtual spaces, and (ii) extend MDT routing to optimize end-to-end path costs for any additive routing metric (such as, latency or ETT [14]).

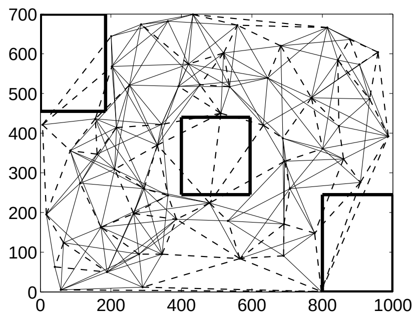
[1]Routing and distance stretch are defined later.



(a) Connectivity graph



(b) DT graph



(c) MDT graph

Figure 2.1: An illustration of connectivity, DT, and MDT graphs of a set of nodes in 2D

MDT was designed to leverage the guaranteed delivery property of Delaunay triangulation (DT) graphs. For nodes in 2D, Bose and Morin proved that greedy routing in a DT always finds a given destination node [7]. Lee and Lam [30] generalized their result and proved that in a d-dimensional Euclidean space ($d \geq 2$), given a destination location ℓ, greedy routing in a DT always finds a node that is closest to ℓ.

Figure 2.1(a) shows a 2D space with three large obstacles and an arbitrary connectivity graph. Figure 2.1(b) shows the DT graph [19] of the nodes in Figure 2.1(a). In the DT graph, the dashed lines denote DT edges between nodes that are not connected by physical links. The MDT graph of the connectivity graph in Figure 2.1(a) is illustrated in Figure 2.1(c). By definition, the MDT graph includes every physical link in the connectivity graph and every edge in the DT graph. In MDT routing, when a packet is stuck at a local minimum of the connectivity graph. the packet is next forwarded, via a “virtual link,” to the DT neighbor that is closest to the destination. In short, the recovery method of MDT is to forward greedily in the DT graph which is guaranteed to succeed.

In this chapter, we present MDT protocols for a set of nodes to construct and maintain a *correct multi-hop DT* (formal definition in Section 2.1). In a multi-hop DT, two nodes that are neighbors in the DT graph communicate directly if there is a physical link between them; otherwise, they communicate via a *virtual link*, i.e., a multi-hop path provided by soft-state forwarding tables in nodes along the path.

MDT protocols are also designed specially for networks where node churn and link churn are nontrivial concerns.

The MDT protocol suite consists of protocols for forwarding, join, leave, failure, maintenance, and system initialization. The MDT join protocol was proved correct for a single join. Thus it constructs a correct multi-hop DT when nodes join serially. The maintenance protocol enables concurrent joins at system initialization. Experimental results show that MDT constructs a correct multi-hop DT very quickly using concurrent joins. The join and maintenance protocols are sufficient for a system under churn to provide a routing success rate close to 100% and for node states to converge to a correct multi-hop DT after churn. The leave and failure protocols are used to improve accuracy and reduce communication cost.

MDT is communication efficient because MDT does not use flooding to discover multi-hop DT neighbors. MDT’s search technique is also not limited by a maximum hop count (needed in scoped flooding used by many wireless routing protocols) and is guaranteed to succeed when the existing multi-hop DT is correct.

For a given set of nodes, under the restrictive assumption that every node can directly communicate with every other node, Lee and Lam [30, 32] presented protocols for the construction and maintenance of a correct distributed DT. These protocols cannot be used for layer-2 routing because the assumption is not satisfied. Major contributions of this work include the definition of a correct distributed multi-hop DT, a new two-step greedy forwarding protocol, proofs of guaranteed delivery by the new forwarding protocol and correctness of the join protocol, as well as designing each protocol in the MDT suite to *correctly* construct/repair forwarding tables in paths between multi-hop DT neighbors to provide a correct distributed multi-hop DT.

2.1 Concepts and Definitions

A triangulation of a set S of nodes (points) in 2D is a maximal planar subdivision[2] of the convex hull of nodes in S into non-overlapping triangles such that the vertices of each triangle are nodes in S. A DT in 2D is a triangulation such that the circumcircle of each triangle does not contain any other node inside [19]. The definition of DT can be generalized to a higher dimensional space using simplexes and circum-hyperspheres. In each case, the DT of S is a graph to be denoted by $DT(S)$.

Consider a set S of nodes in a d-dimensional space, for $d \geq 2$. Each node in S is identified by its location specified by coordinates. There is at most one node at each location. When we say node u *knows* node v, node u knows node v's coordinates. A node's coordinates may be accurate, inaccurate, or arbitrary (that is, its known location may differ from its actual location). In Section 2.1.1, we present the definition of a *distributed* DT and a key result from Lee and Lam [30, 32] that we need later.

2.1.1 Distributed DT

A distributed DT of a set S of nodes is specified by $\{< u, N_u > | u \in S\}$, where N_u represents the set of u's neighbor nodes, which is locally determined by u.

Definition 1. A distributed DT is **correct** if and only if for every node $u \in S$, N_u is the same as the neighbor set of u in $DT(S)$.

[2]A maximal planar subdivision is a subdivision such that no more edge connecting two nodes can be added to the subdivision without destroying its planarity.

To construct a correct distributed DT, each node, $u \in S$, discovers a set C_u of nodes (C_u includes u). Then u computes $DT(C_u)$ locally to determine its set N_u of neighbor nodes. Note that C_u is information discovered by u while S is global knowledge. For the extreme case of $C_u = S$, u is guaranteed to know its neighbors in $DT(S)$. However, the communication cost for each node to discover S (using, for example, a broadcast protocol) would be very high and not scalable. A *necessary and sufficient condition* [30, 32] for a distributed DT to be *correct* is that for all $u \in S$, C_u includes all neighbor nodes of u in $DT(S)$. The condition's necessity is obvious. Its sufficiency requires a nontrivial proof (see [32]). This result enabled the design of efficient protocols for distributed DT construction.

2.1.2 Model assumptions

Two nodes directly connected by a physical link are said to be *physical neighbors*. Each link is bidirectional. In our protocol descriptions, each link is assumed to provide reliable message delivery.[3]

The graph of nodes and physical links may be arbitrary so long as it is a connected graph. We provide protocols to handle dynamic topology changes. In particular, new nodes may join and existing nodes may leave or fail.[4] Furthermore, new physical links may be added and existing physical links that have become error-prone are deleted.

Each node runs the same protocols. After a node boots up, it knows all of its physical neighbors. Subsequently, it discovers other nodes, including its multi-hop DT neighbors, from sending and receiving protocol messages.

2.1.3 Multi-hop DT

A multi-hop DT is specified by $\{< u, N_u, F_u > | u \in S\}$, where F_u is a soft-state forwarding table, and N_u is u's neighbor set which is derived from information in F_u. The multi-hop DT model generalizes the distributed DT model by relaxing the requirement that every node in

[3]Only links that are reliable and have an acceptable error rate are included in the connectivity graph.
[4]When a node fails, it becomes silent.

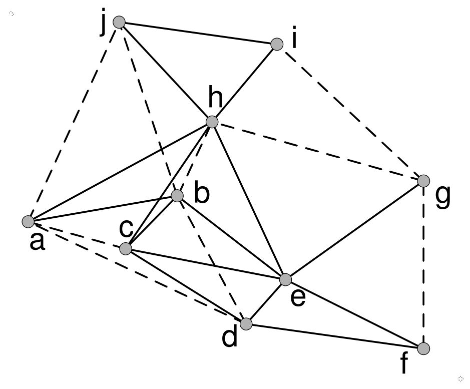


Figure 2.2: MDT graph of 10 nodes

S be able to communicate directly with each of its neighbors. (In this chapter we use the term "neighbor" to refer to a DT neighbor.) In a multi-hop DT, the neighbor of a node may not be a physical neighbor; see, for example, nodes i and g in Figure 2.2.

For a node u, each entry in its forwarding table F_u is a 4-tuple $< source, pred, succ, dest >$, which is a sequence of nodes with *source* and *dest* being the source and destination nodes of a path, and *pred* and *succ* being node u's predecessor and successor nodes in the path. In a tuple, *source* and *pred* may be the same node; also, *succ* and *dest* may be the same node. A tuple in F_u is used by u for message forwarding from *source* to *dest* or from *dest* to *source*. For a specific tuple t, we use $t.source$, $t.pred$, $t.succ$, and $t.dest$ to denote the corresponding nodes in t.

For ease of exposition, we assume that a tuple and its "reverse" are inserted in and deleted from F_u as a pair. For example, $< a,b,c,d >$ is in F_u if and only if $< d,c,b,a >$ is in F_u. (In fact, only one tuple is stored with each of its two endpoints being both source and destination.) A tuple in F_u with u itself as the source is represented as $< -,-, succ, dest >$, which does not have a reverse in F_u.

For an example of a forwarding path, consider the MDT graph in Figure 2.2. The DT edge between nodes g and i is a virtual link; messages are routed along the paths, $g-e-h-i$ and $i-h-e-g$, using the following tuples: $< -,-,e,i >$ in node g, $< g,g,h,i >$ in node e, $< g,e,i,i >$ in node h, and $< -,-,h,g >$ in node i.

Tuples in F_u are maintained as **soft states**. Each tuple is *refreshed* whenever there is packet traffic (e.g., application data or keep-alive messages) between its endpoints. A tuple that is not refreshed will be deleted when its timeout occurs.

Definition 2. A multi-hop DT of S, $\{< u, N_u, F_u > | u \in S\}$, is **correct** if and only if the following conditions hold: i) the distributed DT of S, $\{< u, N_u > | u \in S\}$, is correct; and ii) for every neighbor pair (u, v), there exists a unique k-hop path between u and v in the forwarding tables of nodes in S, where k is finite.

For a dynamic network in which nodes and physical links may be added and deleted, we define a metric for quantifying the accuracy of a multi-hop DT. We consider a node to be *in-system* from when it has finished joining until when it starts leaving or has failed. Let $MDT(S)$ denote a multi-hop DT of a set S of in-system nodes. Let $N_c(MDT(S))$ be the total number of correct neighbor entries and $N_w(MDT(S))$ be the total number of wrong neighbor entries in the forwarding tables of all nodes. A neighbor v in N_u is correct when u and v are neighbors in $DT(S)$ and wrong when u and v are not neighbors in $DT(S)$. Let $N_{edges}(DT(S))$ be the number of edges in $DT(S)$. Let $N_{np}(MDT(S))$ be the number of edges in $DT(S)$ that do not have forwarding paths in the multi-hop DT of S. The **accuracy** of $MDT(S)$ is defined to be:

$$\frac{N_c(MDT(S)) - N_w(MDT(S)) - 2 \times N_{np}(MDT(S))}{2 \times N_{edges}(DT(S))} \tag{2.1}$$

It is straightforward to prove that the accuracy of $MDT(S)$ is 1 (or 100%) if and only if the multi-hop DT of S is correct.

Terminology. For a node u, a physical neighbor v that has just booted up is represented in F_u by the tuple $< -, -, -, v >$. A physical neighbor v that has sent a join request and received a join reply from a DT node is said to be a *physical neighbor attached to the DT*. It is represented in F_u by $< -, -, v, v >$. We use P_u to denote u's set of physical neighbors attached to the DT. A node in P_u will become a DT node when it finishes executing the join protocol.

2.2 MDT Forwarding Protocol

The key idea of MDT forwarding at a node, say u, is conceptually simple: For a packet with destination d, if u is not a local minimum, the packet is forwarded to a physical neighbor closest to d; else, the packet is forwarded, via a virtual link, to a multi-hop DT neighbor closest to d.

For a more detailed specification, consider a node u that has received a data message m to forward. Node u stores it with the format: $m =< m.dest, m.source, m.relay, m.data >$ in a local data structure, where $m.dest$ is the destination location, $m.source$ is the source node, $m.relay$ is the relay node, and $m.data$ is the payload of the message. Note that if $m.relay \neq null$, message m is traversing a virtual link.

Table 2.1: MDT forwarding protocol at node u

	CONDITION	**ACTION**
1.	$u = m.dest$	no need to forward (node u is at destination location)
2.	there exists a node v in P_u and $v = m.dest$	transmit to v (node v is at destination location)
3.	$m.relay \neq null$ and $m.relay \neq u$	find tuple t in F_u with $t.dest = m.relay$, transmit to $t.succ$
4.	there exists a node v in $P_u \cup \{u\}$ closest to $m.dest$, $v \neq u$	transmit to node v (greedy step 1)
5.	there exists a node v in $N_u \cup \{u\}$ closest to $m.dest$, $v \neq u$	find tuple t in F_u with $t.dest = v$, transmit to $t.succ$ (greedy step 2)
6.	conditions 1-5 are all false	no need to forward (node u is closest to destination location)

The MDT forwarding protocol at a node, say u, is specified by the conditions and actions in Table 2.1. To forward message m to a node closest to location $m.dest$, the conditions in Table 2.1 are checked sequentially. The first condition found to be true determines the forwarding action. In particular, line 3 is for handling messages traversing a virtual link. Line 4 is greedy forwarding to physical neighbors. Line 5 is greedy forwarding to multi-hop DT neighbors.

The following theorem states that MDT forwarding in a correct multi-hop DT provides guaranteed delivery.

Theorem 1 *Consider a correct multi-hop DT of a finite set S of nodes in a d-dimensional Euclidean space ($d \geq 2$). Given a location ℓ in the space, the MDT forwarding protocol succeeds to find a node in S closest to ℓ in a finite number of hops.*

Proof:

1) By definition, a correct multi-hop DT of S is a correct distributed DT of S. The distributed DT maintained by nodes in S is the same as $DT(S)$.

2) Given a correct multi-hop DT, each DT neighbor of a node u in S is either a physical neighbor or connected to u by a forwarding path of finite length (in hops) that exists in $\{F_v \mid v \in S\}$.

3) When a message, say m, arrives at a node, say u, if the condition in line 1, 2, or 6 in Table 2.1 is true, then a node closest to ℓ is found. If the conditions in lines 1-3 are all false, node u performs greedy forwarding in lines 4-5. If it succeeds to find in P_u a physical neighbor v that is closer to ℓ than node u, message m is transmitted directly to v (lines 4 in Table 2.1); else, greedy forwarding is performed over the set of DT neighbors (line 5 in Table 2.1). The proof of Theorem 1 in [30] for a distributed DT guarantees that either node u is closest to ℓ or there exists in N_u a node v that is closer to ℓ than u. Therefore, if node u is not a closest node to ℓ, executing the *greedy forwarding code* (lines 4-5 in Table 2.1) finds a node v that is closer to ℓ than node u.

4) Any other node in S that is closer to ℓ than u will not use the actions in lines 4-5 in Table 2.1 to send message m back to node u. It is, however, possible for message m to visit node u again in the forwarding path between two DT neighbors that are closer to ℓ than u. In this case, the condition of line 3 in Table 2.1 must be true for m at node u. Thus, node u executes the greedy forwarding code (lines 4-5 in Table 2.1) for message m at most once. This property holds for every node. By 2), 3), and the assumption that

S has a finite number of nodes, MDT forwarding finds a closest node in S to ℓ in a finite number of hops. $\square$

2.3 MDT Protocol Suite

In addition to the forwarding protocol, MDT includes *join*, *maintenance*, *leave*, *failure*, and *initialization* protocols. The join protocol is designed to have the following correctness property: Given a system of nodes maintaining a correct multi-hop DT, after a new node has finished joining the system, the resulting multi-hop DT is correct. This property ensures that a correct multi-hop DT can be constructed for any system of nodes by starting with one node, say u with $F_u = \emptyset$ initially, which is a correct multi-hop DT by definition, and letting the other nodes join the existing multi-hop DT serially.

Two nodes are said to join a system *concurrently* if their join protocol executions overlap in time. When two nodes join concurrently, the joins are *independent* if the sets of nodes whose states are changed by the join protocol executions do not overlap. For a large network, two nodes joining different parts of the network are likely to be independent. If nodes join a correct multi-hop DT concurrently and independently using the MDT join protocol, the resulting multi-hop DT is also guaranteed to be correct.

The maintenance protocol is designed to repair errors in node states after concurrent joins that are dependent, after nodes leave or fail, after the addition of physical links, and after the deletion of existing physical links (due to, for example, degraded link quality). Experimental results show that join and maintenance protocols are sufficient for a system of nodes to recover from dynamic topology changes and their multi-hop DT to converge to 100% accuracy.

MDT includes leave and failure protocols designed for a single leave and failure, respectively, for two reasons: (i) A departed node has almost all recovery information in its state to inform its neighbors how to repair their states. Such recovery information is not available to the maintenance protocol and would be lost if not provided by a leave or failure protocol before the node leaves or fails. (For failure recovery, each node *u* pre-stores the recovery information in a selected neighbor which serves as *u*'s monitor node.) Thus having leave and failure protocols allows the maintenance protocol, which has a higher communication cost, to run less frequently than otherwise. (ii) Concurrent join, leave, and failure occurrences in different parts of a large network are often independent of each other. After a leave or failure, node states can be quickly and effectively repaired by leave and failure protocols without waiting for the maintenance protocol to run.

For a multi-hop DT, in addition to constructing and maintaining a distributed DT, join and maintenance protocols insert tuples into forwarding tables and update some existing tuples to *correctly* construct paths between multi-hop neighbors. Leave, failure, and maintenance protocols construct a new path between two multi-hop neighbors whenever the previous path between them has been broken due to a node leave/failure or a link deletion.

2.3.1 Join protocol

Consider a new node, say *w*. It boots up and discovers its physical neighbors. If one of the physical neighbors is a DT node (say *v*) then *w* sends a join request to *v* to join the existing DT.[5] In the MDT join protocol, a node uses the basic search technique of Lee and Lam [30] to find its DT neighbors. First, greedy forwarding of *w*'s join request finds *w*'s closest DT neighbor. Subsequently, *w* sends a neighbor-set request to every new neighbor it has found; each new neighbor replies with a set of *w*'s neighbors in the new neighbor's *local* view. If more new neighbors are found in the replies, *w* sends a neighbor-set request to each of them. This search process is iteratively repeated until *w* finds no more new neighbor in the replies. The MDT join protocol also constructs a forwarding path between *w* and every one of its multi-hop DT neighbors. A more detailed protocol description follows.

Finding the closest node and path construction. Node *w* joins by sending a join request to node *v* with *w*'s own location as the destination location. MDT forwarding is

[5]If node *w* discovers only physical neighbors, it will not start the join protocol until it hears from a physical neighbor that is attached to the DT, e.g., it receives a token from such a node at system initialization.

used to forward the join request to a DT node z that is closest to w (success is guaranteed by Theorem 1). A forwarding path between w and z is constructed as follows. When w sends the join request to v, it stores the tuple $< -, -, v, v >$ in its forwarding table. Subsequently, suppose an intermediate node (say u) receives the join request from a physical neighbor (say v) and forwards it to a physical neighbor (say e), the tuple $< w, v, e, e >$ is stored in F_u.

When node z finally receives the join request of w from a physical neighbor (say d), it stores the tuple $< -, -, d, w >$ in its forwarding table for the reverse path. The join reply is forwarded along the reverse path from z to w using tuples stored when the join request traveled from w to z earlier. Additionally, each such tuple is updated with z as an endpoint. For example, suppose node x receives a join reply from z to w from its physical neighbor e. Node x changes the existing tuple $< e, e, *, w >$ in F_x to $< z, e, *, w >$, where $*$ denotes any node already in the tuple.

After node w has received the join reply, it notifies each of its physical neighbors that w is now attached to the DT and they should change their tuple for w from $< -, -, -, w >$ to $< -, -, w, w >$.

Physical-link shortcuts. The join reply message, at any node along the path from z to w (including node z), can be transmitted directly to w if node w is a physical neighbor (i.e., for message m, there is a tuple t in the forwarding table such that $t.dest = m.dest$). If such a physical-link shortcut is taken, the path previously set up between z and w is changed. Tuples with z and w as endpoints stored by nodes in the abandoned portion of the previous path will be deleted because they will not be refreshed by the endpoints.

A physical-link shortcut can also be taken when other messages in the MDT join, maintenance, leave, and failure protocols are forwarded, but they require the stronger condition: there is a tuple t in the forwarding table such that $t.succ = t.dest = m.dest$, that is, the shortcut can be taken only if $m.dest$ is a physical neighbor *attached to the DT*.

Finding DT neighbors. Node w, after receiving the join reply from node z, sends a neighbor-set request to z for neighbor information. At this time, C_z, the set of nodes known to z includes both w and z. Node z computes $DT(C_z)$, finds nodes that are neighbors of w in $DT(C_z)$, and sends them to w in a neighbor-set reply message.

When w receives the neighbor-set reply from z, w adds neighbors in the reply (if any) to its candidate set, C_w, and updates its neighbor set, N_w, from computing $DT(C_w)$. If w finds new neighbors in N_w, w sends neighbor-set requests to them for more neighbor information. The joining node w iteratively repeats the above search process until it cannot find any more new neighbor in N_w. At this time w has successfully joined and become a DT node.

Nodes in C_u, the set of nodes known to a node u, are maintained as hard states in distributed DT protocols [30, 32]. In MDT protocols, nodes in C_u are maintained as soft states. More specifically, tuples in F_u are maintained as soft states. By definition, C_u consists of nodes in $\{u\} \cup \{v \mid v = t.dest,\ t \in F_u\}$ as well as new nodes that may become DT neighbors. A new node in C_u is deleted if it does not become the destination of a tuple in F_u within a timeout period. Furthermore, whenever a tuple t is deleted from F_u upon timeout, each endpoint of t is deleted from C_u unless it is also an endpoint of another tuple.

Path construction to multi-hop DT neighbors. The MDT join protocol also constructs a forwarding path between the joining node w and each of its multi-hop neighbors. Whenever w learns a new node y from the join reply or a neighbor-set reply sent by some node, say x, node w sends a neighbor-set request to x, with x as the *relay* and y as the destination (that is, in neighbor-set request m, $m.relay = x$ and $m.dest = y$.) Note that a forwarding path has already been established between w and x. Also, since x and y are DT neighbors, a forwarding path exists between x and y (given that w is joining a correct multi-hop DT). As the neighbor-set request is forwarded and relayed from w to y, tuples with w and y as endpoints are stored in forwarding tables of nodes along the path from w to y. The forwarding path that has been set up between w and y is then used by y to return a neighbor-set reply to w.

Note that $m.relay$ serves two different functions in different types of MDT protocol messages [28]. In a data message (also a join request message), *m.relay* is used to indicate a multi-hop DT neighbor that can route the message out of a local minimum. In a neighbor-set request message sent by a joining node (say u), *m.relay* is the node that previously informed u of the new node to which the neighbor-set request is destined.

Example. Let node a in Figure 2.2 be a joining node. Suppose a has found b, c, and d to be DT neighbors and it has just learned from b that j is a new neighbor. Node a sends a neighbor-set request to j with b indicated in the message as the relay. Because the existing multi-hop DT (of 9 nodes) is correct, a unique forwarding path exists between node b and node j, which is $b-e-h-j$. After receiving the message, b forwards it to e on the $b-e-h-j$ path. At b and every node along the way to j, a tuple with endpoints a and j is stored in the node's forwarding table. When the neighbor-set reply from j travels back via h, node h searches F_h and finds that node a is a physical neighbor attached to the DT (see Figure 2.2). Node h then transmits j's reply directly to node a. (This is an example of a *physical-link shortcut*.) Subsequently, nodes a and j will select and refresh only the path $a-h-j$ between them. Tuples previously stored in nodes b, e, and h for endpoints a and j will be deleted upon timeout. Lastly, from j's reply, a learns no new neighbor other than b, c, and d. Without any more new neighbor to query, a's join protocol execution terminates and it becomes a DT node.

Theorem 2 *Let S be a set of nodes and w be a joining node that is a physical neighbor of at least one node in S. Suppose the existing multi-hop DT of S is correct, w joins using the MDT join protocol, and no other node joins, leaves, or fails. Then the MDT join protocol finishes and the updated multi-hop DT of $S \cup \{w\}$ is correct.*

Proof: By Theorem 1, the join request of w succeeds to find a DT node (say z) closest to w, which sends back a joint reply. By a property of DT, node z, being closest to w, is guaranteed to be a neighbor of w in $DT(S \cup \{w\})$. A forwarding path is constructed between w and z. Subsequently, because the multi-hop DT of S is correct, forwarding paths are constructed between w and each neighbor it sends a neighbor-set request. After receiving a request from w, each neighbor of w updates its own neighbor set to include w. They also send back replies to w. By Lemma 9 in [30], the join process finishes and N_w consists of all neighbor nodes of w in $DT(S \cup \{w\})$.

By construction, two DT neighbors select only one path to use between them by refreshing only tuples stored in nodes along the selected path. Therefore, the path between each pair of neighbors in $DT(S \cup \{w\})$ is unique after the join. Each path also has a finite number of hops because (i) the path from the joining node to its closest DT node (z) has a finite number of hops (by Theorem 1), and (ii) the path from the joining node to each of its other DT neighbors is either a one-hop path or the concatenation of two paths, each of which has a finite number of hops. By Definition 3, the updated multi-hop DT is correct. $\square$

In the above proof, we make use of Lemma 9 in [30] for a distributed DT in which every node can directly communicate with every other node. Let S' denote $S \cup \{w\}$. The main ideas used in the proof of this lemma are the following: i) The existing node $z \in S$ closest to the joining node w is a neighbor of w in $DT(S')$. ii) For any two neighbors of w in $DT(S')$, say u and v, if the facet shared by the Voronoi cells of u and w is adjacent to the facet shared by the Voronoi cells of v and w in $DT(S')$, then u and v are neighbors in $DT(S)$. Therefore, having found at least one neighbor in $DT(S')$, w can find any other neighbor in $DT(S')$ by following a sequence of existing edges in $DT(S)$. A detailed proof is presented in [30].

2.3.2 Maintenance protocol

The MDT maintenance protocol for repairing node states is designed for systems with frequent addition and deletion of nodes and physical links. For a distributed DT to be correct, each node must know all of its neighbors in the global DT. Towards this goal, each node (say u) runs the maintenance protocol by first querying a subset of its neighbors, one for each simplex including u in $DT(C_u)$. More specifically, node u selects the smallest subset V of neighbors such that every simplex including u in $DT(C_u)$ includes one node in V. Node u then sends a neighbor-set request to each node in V. A node z that has received the neighbor-set request adds u to C_z and computes $DT(C_z)$. Node z then sends a neighbor-set reply containing neighbors of u in $DT(C_z)$ to u.

Node u adds new nodes found in each neighbor-set reply to C_u; it then computes $DT(C_u)$ to get N_u. If u finds a new neighbor, say x, in N_u, node u sends a neighbor-set request to x if x satisfies the following condition:[6]

C1. The simplex in $DT(C_u)$ that includes both u and neighbor x does not include any node to which u has sent a neighbor-set request.

Node u keeps sending neighbor-set requests until it cannot find any more new neighbor in N_u that satisfies **C1**. Node u then sends neighbor-set notifications to neighbors in N_u that have not been sent neighbor-set requests (these notifications announce u's presence and do not require replies). The protocol code for constructing forwarding paths between node u and each new neighbor is the same as in the MDT join protocol.

If after sending a neighbor-set request to a node, say v, and a neighbor-set reply is not received from v within a timeout period, node v is deemed to have failed. Node u sends a failure notification about v to inform each node in u's updated neighbor set. These notifications are unnecessary since MDT uses soft states; they are performed to speed up convergence to correct node states.

Each node runs the maintenance protocol independently, controlled by a timeout value T_m. After a node has finished running the maintenance protocol, it waits for time T_m before starting the maintenance protocol again. The value of T_m should be set adaptively. When a system has a low churn rate, a large value should be used for T_m to reduce communication cost. We found that if each node runs the maintenance protocol repeatedly, the node states converge to a correct multi-hop DT very quickly. (See results from our system initialization experiments in Section 2.4.3 and churn experiments in Section 2.4.6.)

[6]The maintenance protocol can use the same iterative search technique used in the join protocol. However, experimental results show that condition **C1** can be used to reduce the number of neighbor-set messages sent by the maintenance protocol without any impact on its effectiveness to find DT neighbors.

2.3.3 Initialization protocols

Serial joins by token passing. Starting from one node, other nodes join serially using the join protocol. The ordering of joins is controlled by the passing of a single token from one node to another.

Concurrent joins by token broadcast. Starting from one node, other nodes join concurrently using the join and maintenance protocols. The ordering of joins is controlled by a token broadcast protocol. Initially, a token is installed in a selected node. When a node has a token, it runs the join protocol once (except the selected node) and then the maintenance protocol repeatedly, controlled by the timeout value T_m. It also sends a token to each physical neighbor that is not known to have joined the multi-hop DT. Each token is sent after a random delay uniformly distributed over time interval $[1, \tau]$, where τ is in seconds. If a node receives more than one token, any duplicate token is discarded.

2.4 Performance Evaluation

2.4.1 Methodology

We evaluate MDT protocols using a packet-level discrete-event simulator in which every protocol message created is routed and processed hop by hop from its source to destination. We will not evaluate metrics that depend on congestion, e.g., end-to-end throughput and latency. Hence, queueing delays at a node are not simulated. Instead, message delivery times from one node to the next are sampled from a uniform distribution over a specified time interval. Time-varying network link characteristics and interference problems are modeled by allowing physical links to be added and deleted dynamically.

Creating general connectivity graphs. To create general connectivity graphs for simulation experiments, a physical space in 3D (2D) is first specified. ***Obstacles*** are then placed in the physical space. The number, location, shape, and size of the obstacles are constrained by the requirement that the unoccupied physical space is not disconnected by the obstacles. (Any real network environment can be modeled accurately if computational cost is not a limiting factor.) ***Nodes*** are then placed randomly in the unoccupied physical space. Let R denote the radio transmission range. ***Physical links*** are then placed using the following algorithm: For each pair of nodes, *if* the distance between them is larger than R or the line between them intersects an obstacle, there is no physical link; *else* a physical link is placed between the nodes with probability p. We refer to p as the *connection probability* and $1-p$ as the *missing link* probability. If a graph created using the above procedure is disconnected, it is not used. Alternatively, to replicate the connectivity graph of a real network, missing links between neighbors can be specified deterministically rather than with probability $1-p$.

Inaccurate coordinates. In this chapter, the coordinates of nodes are simply physical locations, unless otherwise specified. We will discuss virtual coordinates in detail in the next chapter. The known coordinates of a node may be highly inaccurate [38] because some localization methods have large location errors. In our experiments, after placing nodes in the physical space, their "known" coordinates are then generated with randomized location errors. The location errors are generated to satisfy a *location error ratio*, e, which is defined to be the ratio of the average location error to the average distance between nodes that are physical neighbors. We experimented with location error ratios from 0 to 2.

Definitions. The routing stretch value of a pair of nodes, s and d, in a multi-hop DT of S is defined to be the ratio of the number of physical links in the MDT route to the number of physical links in the shortest route in the connectivity graph between s and d. The ***routing stretch*** of the multi-hop DT is defined to be the average of the routing stretch values of all source-destination pairs in S. The ***distance stretch*** of the multi-hop DT is defined similarly with distance replacing number of physical links as metric.

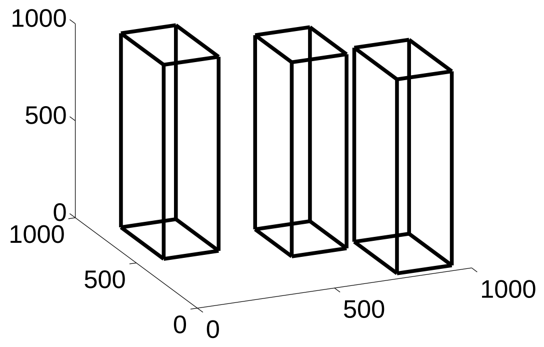


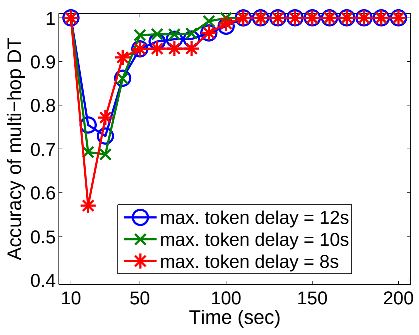
Figure 2.3: Obstacles in a 3D space

2.4.2 Design of experiments

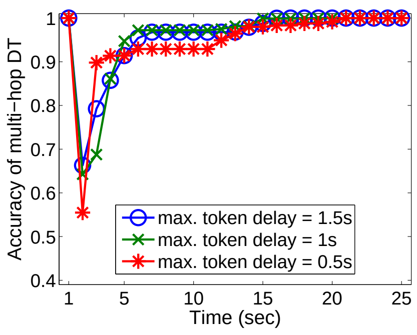
Our simulation experiments were designed to evaluate geographic routing in the most challenging environments. In general, everything else being equal, the challenge is bigger for a higher dimensional space, larger obstacles, a higher missing link probability, a lower node density, a larger network size, or larger node location errors. Furthermore, we performed experiments to evaluate MDT's resilience to dynamic topology changes at very high churn rates. In the geographic routing literature, no other protocol has been shown to meet all of these challenges.

Our simulator enables evaluation of geographic routing protocols in the most challenging environments. In the simulator, any connectivity graph can be created to represent any real network environment with obstacles of different shapes and sizes. The connectivity graphs created as described above have properties of real layer-2 networks, unlike unit-disk and unit-ball graphs used in prior work on geographic routing.[7] We experimented with obstacles of different shapes and sizes, and nodes with large location errors or arbitrary coordinates in 2D, 3D, and 4D. In this chapter, we present experimental results for large obstacles, such as those shown in Figure 2.4.2, because large obstacles are more chal-

[7] In a recent paper on 3D routing, unit-ball graphs were still used for simulation experiments [56].



(a) Ave. message delay = 150 ms



(b) Ave. message delay = 15 ms

Figure 2.4: Accuracy of multi-hop DT vs. time for concurrent joins in 3D

lenging to geographic routing than small ones; these very large obstacles may represent tall buildings in an outdoor space or large machinery in a factory. Between neighbors that are in line of sight and within radio transmission range, we experimented with a missing link probability as high as 0.5.

Node density (average node degree) is an important parameter that impacts geographic routing performance. We present experiments for node densities of 13.5 for 3D and 9.7 for 2D. These node densities are relatively low compared with node densities used in prior work on geographic routing. We found that node densities lower than 13.5 for 3D and 9.7 for 2D would result in many disconnected graphs for spaces with large obstacles and a high missing link probability. We also conducted experiments for higher node densities which resulted in better MDT performance, thus allowing us to conclude that MDT works well for a wide range of node densities. When we scale up the network size in a set of experiments, we increase the space and obstacle sizes to keep node density approximately the same. For experiments with different missing link probabilities, we vary the radio transmission range to keep node density approximately the same.

2.4.3 System initialization experiments

We have performed numerous experiments using our initialization protocols. In every experiment, a correct multi-hop DT is constructed. Concurrent joins can do so much faster than serial joins but with a higher message cost (see Figure 2.11 for message cost comparison).

Figures 2.4(b)-(c) show results from two sets of experiments using concurrent-join initialization. In each experiment, the physical space is a $1000 \times 1000 \times 1000$ 3D space, with three large obstacles, placed as shown in Figure 2.4(a). The size of one obstacle is $200 \times 300 \times 1000$. Each of the other two is $200 \times 350 \times 1000$ in size. The obstacles occupy 20% of the physical space. Connectivity graphs are then created for 300 nodes using the procedure described in Section 2.4.1 for radio transmission range $R = 305$ and link connection probability $p = 0.5$. The *average node degree*, i.e., number of physical neighbors per node, is 13.5. The (known) coordinates of the nodes are inaccurate with location error ratio $e = 1$.

The first set of experiments is for low-speed networks with one-hop message delays sampled from 100 ms to 200 ms (average = 150 ms) and a maintenance protocol timeout duration of 60 seconds. The second set of experiments is for faster networks with one-hop message delays sampled from 10 ms to 20 ms (average = 15 ms) and a maintenance protocol timeout duration of 10 seconds.

In the legend of Figures 2.4(b)-(c), "max. token delay" is maximum token delay τ. In each experiment, note that accuracy of the multi-hop DT is low initially when many nodes are joining at the same time.With a smaller τ, more nodes initiate their join process earlier at about the same time, resulting in a lower MDT accuracy at the beginning. However, accuracy improves and converges to 100% quickly for all τ values. In every experiment, after each node's initial join, the node had run the maintenance protocol only once or twice by the time 100% accuracy was achieved.

2.4.4 MDT performance in 3D

We evaluated the performance of MDT routing for 100 to 1300 nodes in 3D. We present results from four different sets of experiments using connectivity graphs created in a 3D space with and without obstacles, for node locations specified by accurate and inaccurate coordinates. There are four cases:

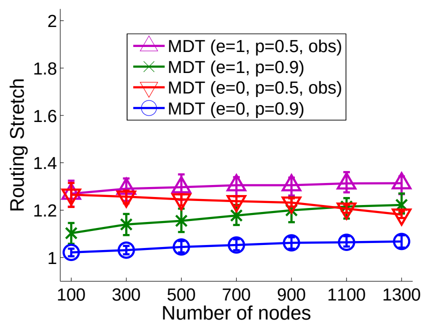
- accurate coordinates ($e = 0$), few missing links ($p = 0.9$), no obstacle
- inaccurate coordinates ($e = 1$), few missing links ($p = 0.9$), no obstacle
- accurate coordinates ($e = 0$), many missing links ($p = 0.5$), large obstacles (*obs*)
- inaccurate coordinates ($e = 1$), many missing links ($p = 0.5$), large obstacles (*obs*)

For 300 nodes, dimensions of the physical space and obstacles are the same as in Figure 2.4(a). For a smaller (or larger) number of nodes, dimensions of the physical space and obstacles are scaled down (or up) proportionally. For each *obs* experiment, the three obstacles are randomly placed in the horizontal plane. $R = 305$ is used for $p = 0.5$ and $R = 250$ is used for $p = 0.9$ such that the average node degree is approximately 13.5. At the beginning of each experiment, a correct multi-hop DT was first constructed. *Routing success rate was 100% in every experiment* and is not plotted.

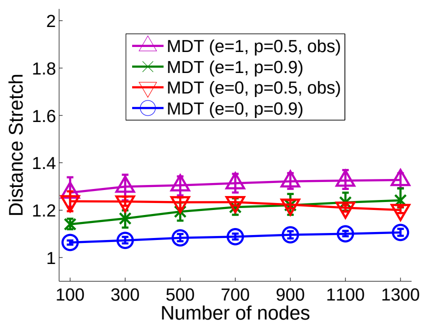
Figures 2.5(a)-(b) show that both routing stretch and distance stretch versus network size are close to 1 for the easy case of accurate coordinates ($e = 0$), few missing links ($p = 0.9$), and no obstacle. Either inaccurate coordinates ($e = 1$) or many missing links ($p = 0.5$) and large obstacles (*obs*) increase both the routing stretch and distance stretch of MDT routing. Note that both the routing and distance stretch of MDT remain low as network size becomes large.[8]

Storage cost. The most important routing information stored in a node is the set of nodes it uses for forwarding; the known coordinates of each node in the set are stored in

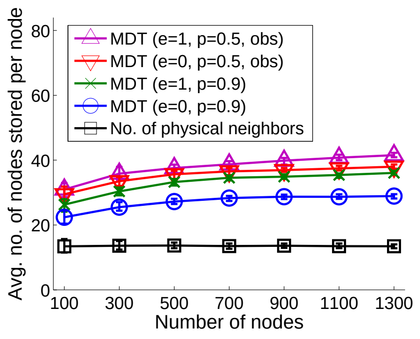
[8]Distance stretch is almost the same as routing stretch (except in 4D experiments for which physical distance is not meaningful) and will not be shown again.



(a) Routing stretch vs. N



(b) Distance stretch vs. N



(c) Storage cost vs. N

Figure 2.5: MDT performance in 3D (average node degree=13.5)

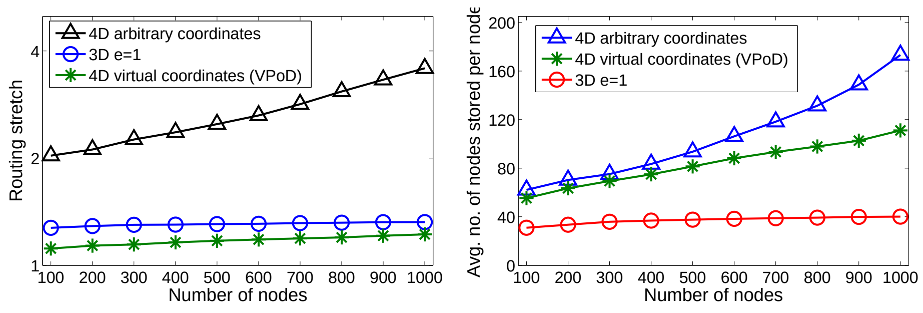
a *location table*. We use 4 bytes per dimension for storing each node's coordinates (e.g., 12 bytes for a node in 3D); this design choice is intended for very large networks. The coordinates of a node are used as its global identifier. Each node is also represented by a 1-byte local identifier in our current implementation. The location table stores pairs of global and local identifiers (e.g., 13 bytes per node for nodes in 3D). In the *forwarding table*, local identifiers are used to represent nodes in tuples. To illustrate MDT's storage cost in bytes, consider the case of 1300 nodes, $e = 1$, and $p = 0.5$ with obstacles. The average location table size is 540.2 bytes. The average forwarding table size is 88.8 bytes. The average location table size is 86% of the combined storage cost. We found that this percentage is unchanged for all network sizes (100 - 1300) in each set of experiments, indicating that the forwarding table size is also proportional to the number of distinct nodes stored.

In this chapter, the storage cost is measured by the average number of distinct nodes a node needs to know (and store) to perform forwarding. This represents the storage cost of a node's minimum required knowledge of other nodes. This metric, unlike counting bytes, requires no implementation assumptions which may cause bias when different routing protocols are compared. Figure 2.5(c) shows the storage cost per node versus network size. As expected, either inaccurate coordinates ($e = 1$) or many missing links ($p = 0.5$) and large obstacles require more storage per node due to the need for more multi-hop DT neighbors. For comparison, the bottom curve is the average number of physical neighbors per node.

Varying obstacle locations. Each data point plotted in Figures 2.5(a)-(c) is the average value of 50 simulation runs for 50 different connectivity graphs each of which was created from a different placement of the obstacles. *Also shown as bars are the 10th and 90th percentile values*. Observe that the intervals between 10th and 90th percentile values are small for all data points. (These intervals are also small in experimental results to be presented in Figures 2.6 and 2.9-2.12 and will be omitted from those figures for clarity.) The small intervals between 10th and 90th percentile values demonstrate that varying obstacle locations has negligible impact on MDT routing performance.

Varying number and size of obstacles. Aside from varying the locations of obstacles, we also experimented with varying the number and size of obstacles. In particular, we repeated the experiments in Figure 2.5 for 6 obstacles and also for 9 obstacles. In each such experiment, the fraction of physical space occupied by obstacles was kept at 20%. We found the resulting changes in MDT's routing stretch, distance stretch, and storage cost to be too small to be visible when plotted in Figures 2.5.[9] However, when we increased the fraction of physical space occupied by obstacles from 20% to 30%, the resulting increases in MDT's routing and distance stretch were significant (about 6%).

[9]Performance measures from experiments for 9 obstacles are smaller than those from experiments for 3 obstacles by less than 0.5%.



(a) Routing stretch vs. N (b) Storage cost vs. N

Figure 2.6: MDT performance in 3D and 4D (average node degree=13.5, $\boldsymbol{p}$=0.5, obstacles)

2.4.5 MDT performance in 4D

To illustrate how MDT can be used in 4D, consider the connectivity graphs created for the set of experiments in Figure 2.5 with many missing links ($p = 0.5$) and large obstacles. Suppose the nodes have *no location information*. We experimented with two cases: (i) Each node assigns itself an arbitrary location in a 4D space and sends its (arbitrary) coordinates to its physical neighbors. These coordinates are used by MDT protocols to construct and maintain a multi-hop DT as well as for routing. (ii) After a multi-hop DT has been constructed by the nodes using the initial (arbitrary) coordinates, each node then runs VPoD [46] which is a virtual positioning protocol that does not require any node location information, any special nodes (such as, landmarks), nor any use of flooding. Nodes use VPoD to change their coordinates by comparing distances with routing costs to their physical and DT neighbors. A new multi-hop DT is then constructed using the new coordinates. After several iterations, the node coordinates will converge to achieve the following property [46]: The distance between any two nodes in the virtual space is a good estimate of the routing cost (*in any additive metric*) between them.

For the results presented in Figure 2.6, we used 1 (hop) as the routing metric between two physical neighbors. Each data point plotted in Figure 2.6 is the average value from 50 experiments.

For comparison, we have also plotted the results for MDT routing using inaccurate

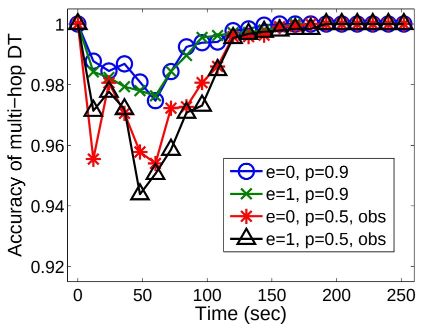
3D coordinates, that is, the case of ($e = 1$, $p = 0.5$, *obs*) in Figure 2.5. Figure 2.6(a) on routing stretch, plotted in logarithmic scale, shows that MDT routing using 4D virtual coordinates is better than using inaccurate coordinates in 3D. Figure 2.6(b) on storage cost shows that MDT routing using inaccurate coordinates in 3D is better than using 4D virtual coordinates. In both figures, MDT routing using arbitrary coordinates has the worst performance. Routing success rate was 100% in every experiment and is not shown.

2.4.6 Resilience to churn

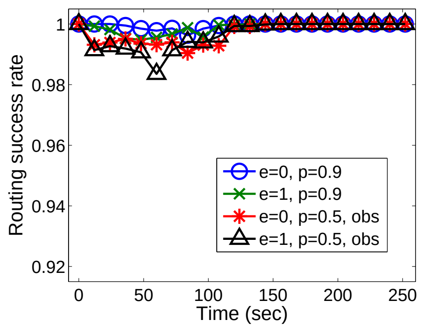
We performed a large number of experiments to evaluate the performance of MDT protocols for systems under churn, with 300 nodes in a $1000 \times 1000 \times 1000$ 3D physical space. Like the experiments used to evaluate MDT routing stretch in Figure 2.5, four sets of experiments were performed using connectivity graphs created with and without three large obstacles, for node locations specified by accurate and inaccurate coordinates. The average node degree is kept at approximately 13.5 for every experiment.

In a *node churn* experiment, the rate at which new nodes join is equal to the churn rate; the rate of nodes leaving and the rate of nodes failing are each equal to half the churn rate. In a *link churn* experiment, the churn rate is equal to the rate at which new physical links are added and the rate at which existing physical links are deleted. In each experiment, the 300 nodes initially maintain a correct multi-hop DT. Churn begins at time=0 and ends at time=60 seconds.

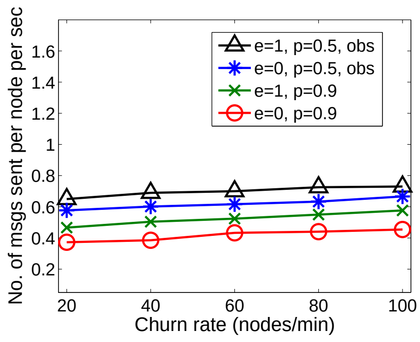
Figure 2.7 presents results from node churn experiments for low-speed networks where one-hop message delays are sampled from [100 ms, 200 ms]. The maintenance timeout value is 60 seconds. The churn rate is 100 nodes/minute in Figures 2.7(a)-(b) and varies in Figure 2.7(c). Figure 2.7(a) shows the accuracy of the multi-hop DT versus time. The accuracy returns to 100% quickly after churn. Figure 2.7(b) shows the routing success rate versus time. The success rate is close to 100% during churn and returns to 100% quickly after churn. Figure 2.7(c) shows the communication cost (per node per second)



(a) Churn rate = 100 nodes/min.



(b) Churn rate = 100 nodes/min.



(c) Communication cost vs. churn rate

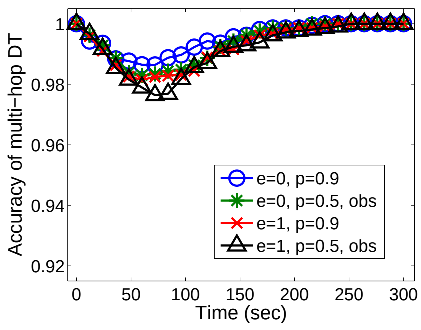
Figure 2.7: MDT performance under node churn (ave. message delay = 150 ms, timeout = 60 sec.)

versus churn rate.

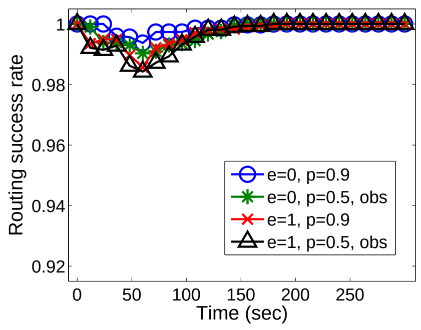
By Little's Law, for 300 nodes and a churn rate of 100 nodes per minute, the average lifetime of a node is 300/100 = 3 minutes, which represents a very high churn rate for most practical systems.

Figure 2.8 presents results from link churn experiments for low-speed networks with a maintenance timeout value of 60 seconds. Figure 2.8(a) shows the accuracy of the multi-hop DT versus time. The accuracy returns to 100% quickly after churn. Figure 2.8(b) shows the routing success rate versus time. The success rate is close to 100% during churn and returns to 100% quickly after churn. Figure 2.8(c) shows the communication cost (per node per second) versus churn rate.

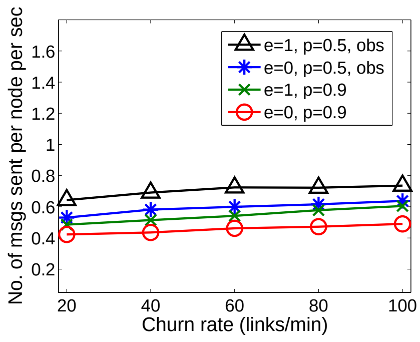
Note that the convergence times to 100% accuracy in Figures 2.7(a) and 2.8(a) and



(a) Churn rate = 100 links/min.



(b) Churn rate = 100 links/min.



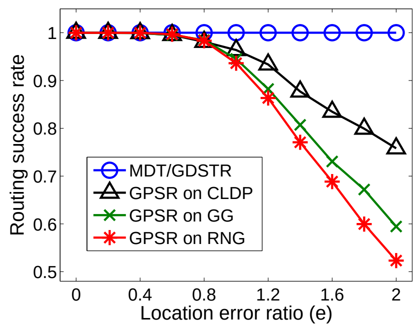
(c) Communication cost vs. churn rate

Figure 2.8: MDT performance under link churn (ave. message delay = 150 ms, timeout = 60 sec.)

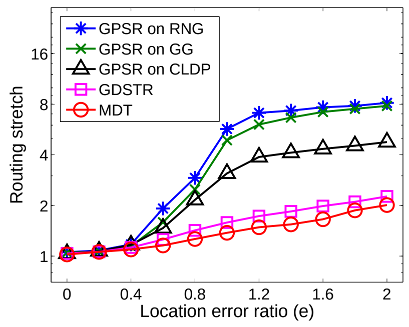
to 100% success rate in Figures 2.7(b) and 2.8(b) are almost the same for the four cases. These results are typical of all churn experiments performed.

2.4.7 Discussion on overheads

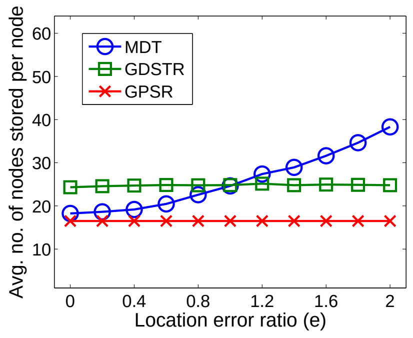
Nodes that implement MDT protocols incur extra storage and communication costs when compared to a simple greedy routing protocol. The extra storage cost of MDT is the difference between the MDT storage cost and the number of physical neighbors; see Figure 2.5(c). Observe that the extra storage cost converges to an asymptotic constant as N becomes large. There are two types of extra communication costs: (i) communication costs to construct a multi-hop DT initially (see Figure 2.11 in the next section); and (ii) communication costs incurred by the maintenance protocol during churn (see Figures 2.7(c) and



(a) Routing success rate vs. *e*



(b) Routing stretch vs. *e*



(c) Storage cost vs. *e*

Figure 2.9: Performance comparison of 2D routing protocols (average node degree=16.5)

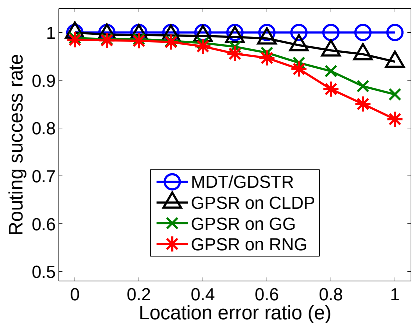
2.8(c)). The per-node churn cost is less than 0.8 message/second for very high churn rates and the most difficult case. Note that when the network topology is static, MDT incurs (essentially) no extra overhead.

2.5 Performance Comparison

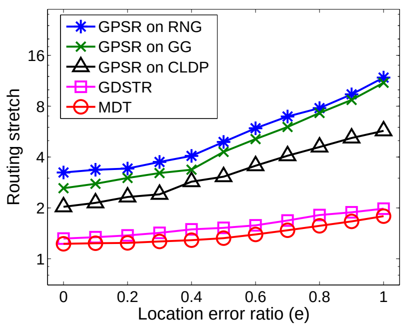
2.5.1 Comparison of 2D protocols

The geographic routing protocols, GPSR running on GG, RNG, and CLDP graphs [24, 27], and GDSTR [35] were designed for routing in 2D. We implemented these protocols in our simulator.[10] The experiments in Figure 2.9 were carried out for 300 nodes in a 1000×1000

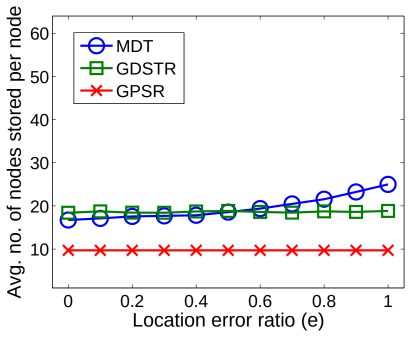
[10]Using, as our references, [27] for CLDP, GDSTR code from www.comp.nus.edu.sg/~bleong/geographic/, and GPSR, GG, and RNG code from www.cs.ucl.ac.uk/staff/B.Karp/gpsr/. GDSTR uses two hull trees [35].



(a) Routing success rate vs. e



(b) Routing stretch vs. e



(c) Storage cost vs. e

Figure 2.10: Performance comparison of 2D routing protocols (three large obstacles, average node degree=9.7)

2D space with no obstacle and few missing links ($p = 0.9$). The radio transmission range is $R = 150$. The average node degree is 16.5. The performance results are plotted versus location error ratio, from $e = 0$ (no error) to $e = 2$ (very large location errors).

The experiments of Figure 2.10 were carried out for 300 nodes in a 1000×1000 2D space with three randomly placed obstacles (a 200×300 rectangle and two 200×350 rectangles) and many missing links ($p = 0.5$). The radio transmission range is $R = 150$. The average node degree is 9.7. The performance results are plotted versus location error ratio, from $e = 0$ to $e = 1$.

In Figure 2.9(a) and Figure 2.10(a) the routing success rates of MDT and GDSTR are both 100% for all e values (it was 100% in every experiment). As the location error ratio (e) increases from 0, the routing success rates of RNG, GG, and CLDP drop off gradually

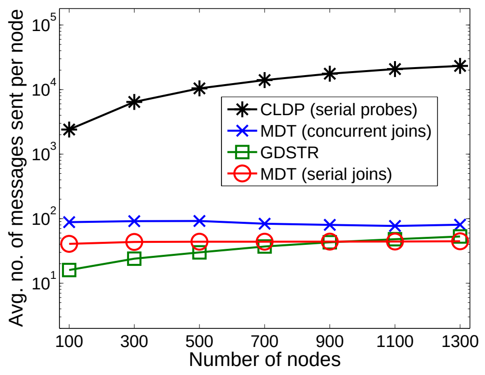


Figure 2.11: Initialization message cost vs. N (average node degree = 12)

from 100%. For $e > 0.6$ in Figure 2.9(a) and $e > 0.3$ in Figure 2.10(a), their routing success rates drop significantly.

Figure 2.9(b) and Figure 2.10(b), in logarithmic scale, show that MDT has the lowest routing stretch for all e values, with GDSTR a close second, followed by CLDP, GG, and RNG in that order. Note that routing stretch increases as e increases for all protocols.

Figure 2.9(c) and Figure 2.10(c) show storage cost comparisons. The GPSR protocols (CLDP, GG, and RNG) have the lowest storage cost, with the storage costs of GDSTR and MDT about the same.

Comparison of graph construction costs. We compare MDT's message cost to construct a correct multi-hop DT with message costs of CLDP graph construction using serial probes [27] and GDSTR hull tree construction [35]. The physical space is a 2D square with three large rectangular obstacles, occupying 20% of the physical space. There are many missing links ($p = 0.5$). Nodes have inaccurate coordinates ($e = 1$). The number N of nodes is varied from 100 to 1300. For the radio transmission range $R = 150$, the sizes of the physical space and obstacles are determined for each value of N such that the average node degree is approximately 12.

In Figure 2.11, the vertical axis is in logarithmic scale. The message cost of a protocol is the average number of messages *sent* per node (we did not account for message size differences among the protocols). Note that each GDSTR message is a *broadcast* message sent by a node to all of its physical neighbors and is counted only as one message sent. Messages sent by CLDP and MDT are unicast messages.

Figure 2.11 shows that with the average number of messages *sent* per node as metric, GDSTR has the best message cost performance for up to 900 nodes. For more than 900 nodes MDT (serial joins) has the lowest cost. CLDP has a very high cost. Note that the CLDP and GDSTR curves increase gradually with N. The MDT curves are flat.

2.5.2 Comparison of 3D protocols

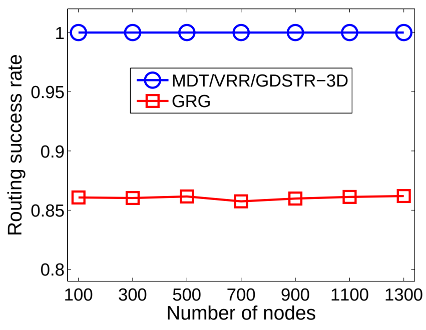
We compare the routing performance of MDT with GRG [17] and GDSTR-3D [56]. We implemented the basic version of GRG in our simulator. Several techniques to improve the performance of GRG are presented for *unit ball graphs* [17]. Since arbitrary connectivity graphs are used in our experiments, these techniques are not applicable and not implemented.

GDSTR-3D uses two hull trees for recovery. For each tree, each node stores two 2D convex hulls to aggregate the locations of all descendants in the subtree rooted at the node; the two 2D convex hulls approximate a 3D convex hull at each node. We implemented GDSTR-3D using its authors' TinyOS 2.x source code available at Google Sites.

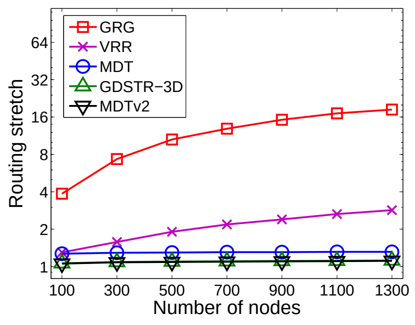
Unlike other geographic protocols, each node in GDSTR-3D stores *2-hop neighbors* and uses 2-hop greedy forwarding to reduce routing stretch at the expense of a much larger storage cost per node. This performance tradeoff may not be appropriate for networks with limited nodal storage.

A non-geographic routing protocol, VRR [9], is included in the comparison. We implemented VRR for static networks without joins and failures.[11] For each pair of virtual neighbors, we used the shortest path (in hops) between them as the forwarding path (the routing stretch value is 1 between virtual neighbors). Thus, the routing stretch and storage

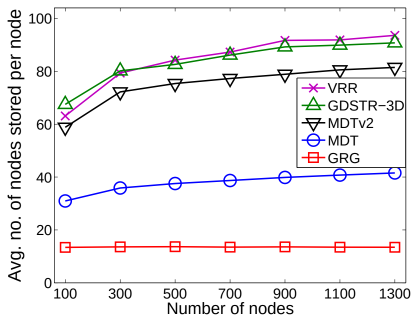
[11] With reference from www.cs.berkeley.edu/~mccaesar/vrrcode .



(a) Routing success rate vs. N



(b) Routing stretch vs. N



(c) Storage cost vs. N

Figure 2.12: Performance comparison of 3D routing protocols (average node degree=13.5)

cost results shown in Figure 2.12(b)-(c) for VRR are slightly optimistic. In VRR, each node also stores 2-hop neighbors for forwarding.

MDT can be easily modified to use 2-hop greedy forwarding. We present results for both MDT (which uses 1-hop greedy forwarding) and MDTv2 (which uses 2-hop greedy fowarding).

In our experiments, the number N of nodes is varied from 100 to 1300. The physical space and large obstacles are the same as the ones used in Figure 2.5. The average node degree was kept at approximately 13.5. Experiments were performed using connectivity graphs created for the following case: inaccurate coordinates ($e = 1$), many missing links ($p = 0.5$), and three large obstacles that occupy 20% of the physical space.

Figure 2.12(a) shows that MDT (also MDTv2), GDSTR-3D, and VRR all achieve

100% routing success rate while the routing success rate of GRG is about 86%. Figure 2.12(b), in logarithmic scale, shows that the routing stretch of GRG is very high, the routing stretch of VRR is high for $N > 300$, and both increase with N. The routing stretch of MDTv2 is the lowest and slightly lower than that of GDSTR-3D for every network size (the differences are, however, too small to be seen in Figure 2.12(b)). MDT, which uses 1-hop greedy forwarding, ranks a close third.

In Figure 2.12(c), GDSTR-3D, VRR, MDTv2 have large per-node storage costs, because each node stores 2-hop neighbors as well as physical neighbors. The storage cost of MDTv2 is smaller than those of GDSTR-3D and VRR. Both GRG and MDT have much lower storage costs because they use 1-hop greedy forwarding. The per-node storage cost of GRG, equal to the average number of physical neighbors, is the lowest of the five protocols.

MDT versus GDSTR-3D. MDT, MDTv2, and GDSTR-3D all provide guaranteed delivery in 3D and achieve routing stretch close to 1. GDSTR-3D has a higher storage cost than MDTv2 and a much higher storage cost than MDT. One clear advantage MDT (or MDTv2) has over GDSTR-3D is that MDT is highly resilient to dynamic topology changes (both node churn and link churn) while GDSTR-3D is designed for a static topology without provision to handle any dynamic topology change. Another advantage of MDT is that it provides guaranteed delivery for nodes with arbitrary coordinates in higher dimensions ($d > 3$).

Chapter 3

Virtual Positioning in layer 2 and Greedy Distance Vector Routing

Distance Vector (DV) is a well-known routing technique. In DV forwarding, a node u chooses a physical (directly connected) neighbor v as the next hop to destination t such that it minimizes the distance $c(u,x)+D(x,t)$ for $x \in P_u$, where $c(u,x)$ is the cost of link u-x, $D(x,t)$ is the least cost from x to t, and P_u is the set of u's physical neighbors. Any *additive* metric can be used for $c(u,x)$ and $D(x,t)$. If nodes have accurate distance vectors, DV routing provides the least cost paths.

Geographic routing uses greedy forwarding as the basis, i.e., a node u selects, as the next hop to destination t, a physical neighbor v that minimizes $d(x,t)$ for $x \in P_u$, where $d(x,t)$ is the physical distance between x and t. Traditionally, geographic routing protocols were evaluated using hop count or physical distance as the routing metric. However, other metrics such as latency, ETX [12], and ETT [14] are also very important in network protocol design and measurement.

In this chapter, we present Greedy Distance Vector (GDV), the first geographic routing protocol designed with the objective of providing near-optimal paths for *any additive routing metric*. To apply GDV, each node assigns itself a virtual position (position in a virtual space) such that the Euclidean distance between any pair of nodes in the virtual space is a good estimate the routing cost between them. Like DV routing, GDV selects as the next hop to destination t, a neighbor v that minimizes $c(u,x) + \tilde{D}(x,t)$ for $x \in P_u$, where $\tilde{D}(x,t)$ is the estimated routing cost from x to t from *locally computing* the distance between the virtual positions of x and t.[1] Since $c(u,v) + \tilde{D}(v,t) \approx c(u,v) + D(v,t)$, the quality of GDV paths is expected to be close to that of optimal DV paths.

Many virtual coordinate schemes have been proposed for wireless networks when node location information is unavailable [48] [18] [10] [39] [55] [36] [54]. Unlike GDV, these schemes were not designed to predict routing costs between nodes. Instead their main objective is to improve greedy delivery rate.

The idea of embedding latencies (routing costs) in a virtual space was used by many virtual positioning systems, such as, GNP [43] and Vivaldi [13]. These systems, however, were designed for hosts with Internet routing support. More specifically, GNP requires that each node makes RTT measurements to a set of *landmark* nodes (some may be far away). Vivaldi requires that each node receives latency measurements from distant nodes from time to time.[2] In layer-2 networks, each node can communicate directly with its physical neighbors. But to communicate with distant nodes, GNP and Vivaldi require any-to-any routing support. Therefore, they cannot be used to support GDV in layer-2 networks.

To illustrate the point that Vivaldi requires routing cost measurements to distant n-odes, consider the 121-node grid network shown in Figure 3.1. Each node is only aware of its local connectivity and has no location information. We enhance the Vivaldi algorithm [13] with routing support such that it can *sample* (measure routing cost to) two-hop neighbors as well as physical neighbors. In each *adjustment period*, a node samples random nodes from its set of one-hop neighbors 100 times and its set of two-hop neighbors 100 times. Figure 3.2 shows the virtual positions of the nodes after 10 and 20 adjustment

[1]For GDV, the neighbor set of node u will be generalized to include both physical and Delaunay triangulation neighbors.

[2]In this chapter, a node is *distant* iff the routing cost to that node is high.

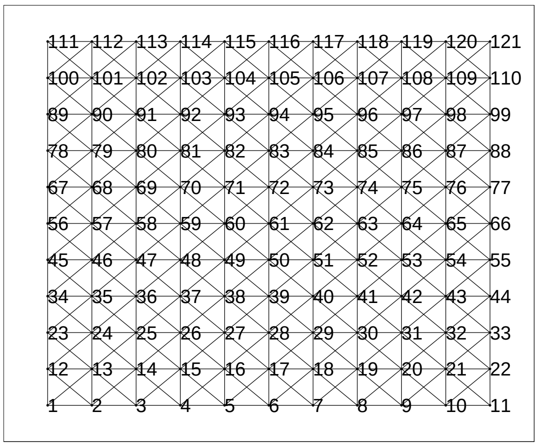


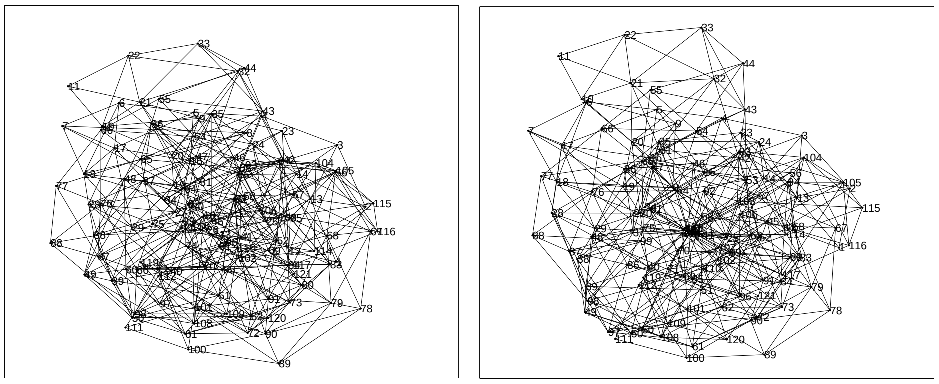
Figure 3.1: 121-node network in 2D physical space

periods (hop count was used as routing metric). We found that almost every node is close to its physical neighbors in the virtual space. However, two nodes that are separated by many hops may also be very close in the virtual space (such as, many of the nodes near the center). Generally, there are two kinds of relationships needed for virtual positions to predict routing costs accurately [13]:

- *Local relationships*: nodes with low cost should be nearby in the virtual space.
- *Global relationships*: nodes with high cost should be far away in the virtual space.

Clearly, in this example, two-hop Vivaldi performs well for local relationships but poorly for global relationships. Routing support to sample two-hop neighbors is not sufficient for Vivaldi to work properly for layer-2 networks.

In this chapter, we present a novel virtual positioning protocol for layer-2 networks, named *Virtual Position by Delaunay* (VPoD). VPoD makes use of the multi-hop DT of a set of nodes located in a Euclidean space [29]. Given a set of nodes running VPoD, the nodes are initially located in a virtual space (2D, 3D, or higher dimension) fairly arbitrarily. Each node u then uses an adjustment algorithm to move its position in the virtual space by comparing $D(u,v)$, its routing cost to node v in the multi-hop DT, with $\tilde{D}(u,v)$, the distance



(a) After 10 adjustment periods (b) After 20 adjustment periods

Figure 3.2: Virtual positions constructed by 2-hop Vivaldi

between *u* and *v* in the virtual space, where node *v* is either a physical neighbor or a multi-hop DT neighbor of *u*. Any additive metric can be used for routing costs. While prior virtual positioning protocols require routing cost information from some distant nodes to be effective, we discovered that VPoD performs very well using just routing costs from a node to its physical and DT neighbors; such cost information can be piggybacked in MDT protocol messages exchanged by neighbors.

The contributions of this work are the following:

- GDV is the *first* geographic routing protocol designed to optimize end-to-end path costs using any *additive routing metric*, in particular, routing metrics that capture network and link characteristics other than physical distances.

- GDV and VPoD are designed for layer-2 networks without location information. Therefore, no localization protocol is needed.

- As a geographic protocol, GDV's storage cost per node remains low as network size (N) increases. Routing cost estimates are computed locally using virtual positions. Unlike DV, there is no need for nodes to exchange distance-vector messages of size $O(N)$.

- VPoD performs very well in preserving both local and global distance relationships between nodes in the virtual space. Every node runs the same protocols. VPoD does not require special nodes, such as, beacons and landmarks, and does not use flooding.
- Experimental results show that GDV and VPoD have good routing performance on both wireline and wireless network topologies.

3.1 Virtual Position Construction

We present the Virtual Position by Delaunay (VPoD) protocol to construct virtual positions for routing cost prediction in a layer-2 network. Each node only knows the link costs to its physical neighbors. The routing metric can be any one that is additive, such as, hop count, latency, ETX, and ETT, etc. Distances in the virtual space and routing costs are measured in the same units. Hence comparison, addition, and subtraction can be operated directly on distances and routing costs.[3]

3.1.1 Main ideas of VPoD

To start VPoD, an arbitrary starting node generates a token and broadcasts the token to the network.[4] Each node with a token runs the VPoD protocol. It first initializes its position in the virtual space by a simple algorithm. It then runs MDT protocols to participate in the construction of a multi-hop DT of the nodes using their locations in the virtual space. We modified the MDT protocols to record routing costs from each node to its multi-hop DT neighbors. Each node then iteratively adjusts its position by checking the positions of its physical and multi-hop DT neighbors to reduce prediction errors. For a node u, VPoD provides two types of adjustments:

[3]Hereafter, whenever we say *distance*, we refer to the Euclidean distance between two nodes in the virtual space rather than the physical distance between the nodes.

[4]The starting node may be selected by any leader election protocol using a simple criterion, e.g., largest ID. Duplicate tokens received by nodes are ignored and do not affect VPoD.

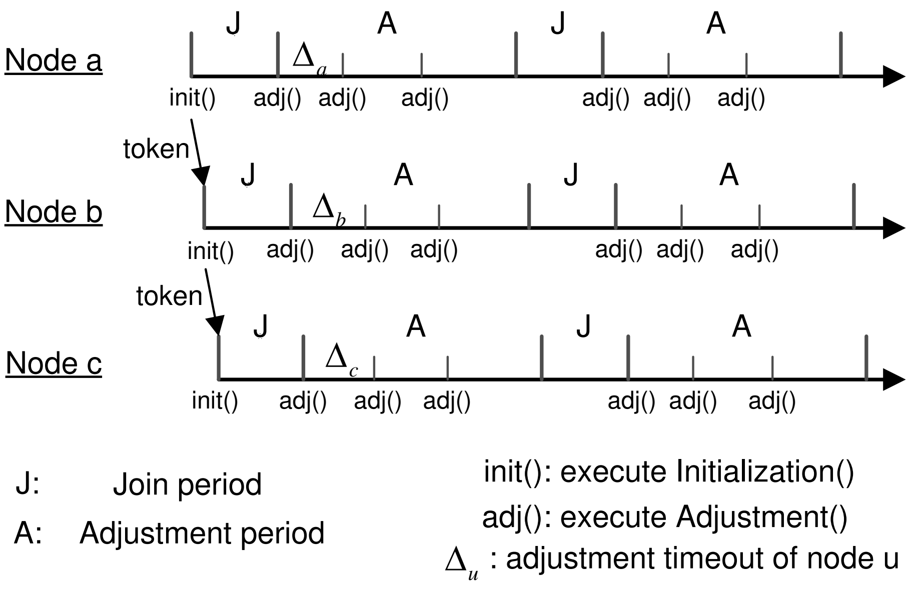


Figure 3.3: Main structure of VPoD

1. Adjustments with physical neighbors to preserve local relationships: If its distance to a physical neighbor v is larger than its link cost to v, u adjusts its position so that its distance to v is smaller.
2. Adjustments with DT neighbors to preserve global relationships: If its distance to a multi-hop DT neighbor v is smaller (larger) than the routing cost from u to v, u adjusts its position so that its distance to v is larger (smaller).

The main structure of VPoD is presented in Figure 3.3. Each node runs the protocol upon receiving a token. After initializing its position, it runs the MDT join and maintenance protocols during a period of time, called *J* period, to participate in constructing the multi-hop DT. In the subsequent adjustment period, called *A* period, the node executes the adjustment algorithm iteratively to change its position in the virtual space. The multi-hop DT needs to be re-constructed after several adjustment iterations because many nodes may have changed their positions. In this manner, each node alternates between execution of the MDT protocols and the adjustment algorithm. The initialization, MDT join and maintenance protocols, and adjustment algorithm will be described in more detail in the sections to follow.

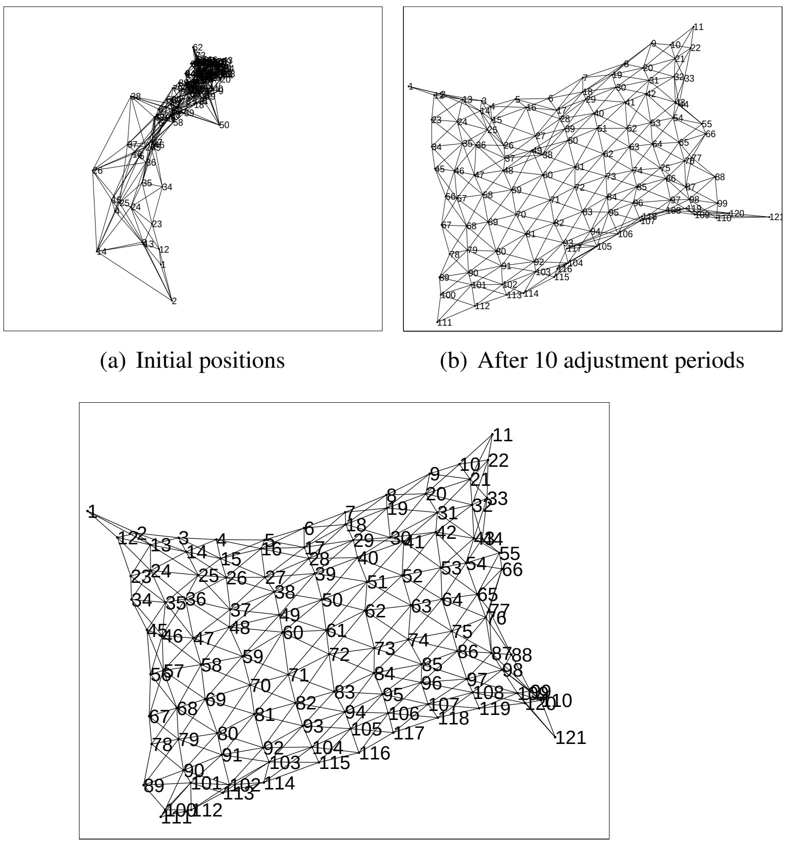
Note that after receiving the initial token, each node runs asynchronously. Different nodes may start their J and A periods at slightly different times. Each node uses its own timer to control the beginning and end of each period. After an adjustment execution, each node sends its new virtual position and estimated error to its physical neighbors and multi-hop DT neighbors. After a number of alternating J and A periods, the node positions in the virtual space converge and distances can be used to predict routing costs between nodes. The MDT protocols are then run one more time to update the multi-hop DT. VPoD does not require any landmark or perimeter node and uses no flooding. Every node in the network runs the same VPoD protocol.

We ran VPoD for the 121-node grid network in Figure 3.1. The results are shown in Figure 3.4. Note that the initial node positions are quite arbitrary. After 10 adjustment periods, the topology in the virtual space looks similar to that in the physical space. After 20 adjustment periods, all local and global relationships are preserved; compare Figure 3.4(c) with Figure 3.1 where nodes are numbered. Note that adjustment periods for VPoD and 2-hop Vivaldi are defined differently. Experimental results in Figure 3.13 (to be presented) show that 2-hop Vivaldi uses much more storage and communication costs per adjustment period than VPoD.

3.1.2 Position initialization

After receiving a token, each node initializes its position in the virtual space before forwarding the token to others. In our current implementation, the initial position of node u is determined as follows.

- If u is the starting node, u sets its position to the origin. Otherwise, at least one physical neighbor of u has initialized its position, namely, the token's sender.
- If only one physical neighbor, say v, of u has initialized its position, u sets its position


(a) Initial positions (b) After 10 adjustment periods

(c) After 20 adjustment periods

Figure 3.4: Virtual positions constructed by VPoD

at a random position on the circle or sphere centered at *v*. The radius is the link cost between *u* and *v*.

- If two or more physical neighbors of *u* have initialized their positions, *u* chooses the two that are farthest apart, and computes the mid-point between the two nodes. In order to avoid degenerate cases (three or more nodes on a line), the actual position of *u* is set to a random position a short distance to the mid-point.

Table 3.1: Notation of VPoD

P_u	physical neighbor set of node u
N_u	DT neighbor set of node u
F_u	forwarding table of node u
x_u	virtual position of node u, a vector
e_u	estimated position error of node u
$\tilde{D}(v,t)$	Euclidean distance between the virtual positions of v and t
$c(u,v)$	cost of the link from u to v
$D(v,t)$	routing cost from node v to node t
Δ_u	adjustment timeout value of node u
c_c, c_e	tuning parameters to control the amounts of change in node position and position error

3.1.3 Multi-hop DT with extensions to support VPoD

We first briefly introduce Delaunay triangulation (DT). A *triangulation* of a set S of nodes (points) in 2D is a subdivision of the convex hull of nodes in S into non-overlapping triangles such that the vertices of each triangle are nodes in S. A DT in 2D is a triangulation such that the circumcircle of each triangle does not contain any other node inside [19]. The definition of DT can be generalized to a higher dimensional Euclidean space using simplexes and circum-hyperspheres. In each case, the DT of S is a graph to be denoted by $DT(S)$.

The model of a *distributed DT* in [31, 32] assumes that each node can directly communicate with every other node in the system. For layer-2 routing, the multi-hop DT model [29] was formulated as an extension of the distributed DT model as follows:

Definition 1: A multi-hop DT is specified by $\{< u, N_u, F_u > | u \in S\}$, where F_u is a soft-state forwarding table and N_u is u's set of DT neighbors locally computed by u from information in F_u.

Definition 2: A multi-hop DT of S, $\{< u, N_u, F_u > | u \in S\}$, is *correct* if and only if the following conditions hold: i) for every node $u \in S$, N_u is the same as the neighbor set of u in $DT(S)$; ii) for every neighbor pair (u,v) in $DT(S)$, there exists a unique k-hop path between *u* and *v* in the forwarding tables of nodes in *S*, where *k* is finite.

A DT neighbor of node *u* may be a physical neighbor if it is directly connected to *u*. If a DT neighbor of *u* is not a physical neighbor, it is said to be a multi-hop DT neighbor. In MDT protocols, each entry in *u*'s forwarding table F_u is a 4-tuple, $< source, pred, succ, dest >$, where *dest* may be a physical or DT neighbor. To meet the requirements of VPoD, each entry in MDT protocols used by VPoD is extended to a 6-tuple $< source, pred, succ, dest, cost, error >$, where *error* is the estimated position error of the *dest* node. If *dest* is a physical neighbor of *u*, *cost* is the link cost to *dest*. If *dest* is a multi-hop DT neighbor, *cost* is the routing cost to *dest*. In tuples where *dest* is neither a physical nor DT neighbor, both *cost* and *error* are empty.

We made another change to MDT join and maintenance protocols to accommodate VPoD as follows. In [29], nodes are globally identified by their coordinates which do not change. In VPoD, each node's virtual position is arbitrary and changes over time. Therefore, nodes in VPoD are identified by globally unique identifiers which are included in MDT messages. However, MDT protocols running under VPoD *do not require a location service* to provide a mapping from global identifiers to virtual positions. Nodes that are physical neighbors exchange messages and learn the updated virtual positions of each other. Also, during execution of the MDT protocols, whenever node *u* learns a new node *x* from node *v*, the message from *v* to *u* includes both the global identifier and virtual position of *x*. When the MDT protocols finish execution, each node knows the global identifiers and virtual positions of all of its physical and DT neighbors.

Additionally, during execution of the MDT join and maintenance protocols, every pair of DT neighbors exchange two messages, *Neighbor_Set_Request* and *Neighbor_Set_Reply*. Each of these messages carries its source node's position error and is also used to record the routing cost of the *reverse path* from its destination node to its source node. When the MDT protocols finish execution, every node knows the *cost* and *error* values of each of its DT neighbors. Also, a path from the node to each of its DT neighbors has been stored in

```
Adjustment():
1.  $e_{\text{sum}} \leftarrow 0$;  // summed error of this adjustment, initialized to 0;
2.  for all $v$ in $P_u \cup N_u$ do
3.    if ($v \in P_u$ and $\tilde{D}(u,v) > D(u,v)$) or $v \in N_u - P_u$ then
4.      $t \leftarrow$ tuple in $F_u$ such that $t.dest = v$;
5.      $e_v \leftarrow t.error$;
6.      $f \leftarrow e_u/(e_u + e_v)$;  // confidence of this update
7.      $x_u \leftarrow x_u + c_c \times f \times [D(u,v) - \tilde{D}(u,v)] \times \hat{u}(x_u - x_v)$;
                    // $\hat{u}(x_u - x_v)$ is a unit vector in the direction of $x_u - x_v$
8.      $e_{\text{sum}} \leftarrow e_{\text{sum}} + |D(u,v) - \tilde{D}(u,v)| / \tilde{D}(u,v)$;
                                        // add the error of this sample
9.    end if
10. end for
11. $e_{\text{new}} \leftarrow e_{\text{sum}}/|P_u \cup N_u|$;  // average error
12. $e_u \leftarrow e_u \times (1 - c_e) + e_{\text{new}} \times c_e$;
13. Send the updated $x_u$ and $e_u$ to all nodes in $P_u \cup N_u$;
```

Figure 3.5: Pseudocode of the VPoD adjustment algorithm at node *u*

forwarding tables of nodes along the path. The *error* values of physical neighbors that are not DT neighbors are exchanged by link-layer keep-alive messages.

Experimental results show that MDT protocols construct a correct multi-hop DT very quickly at system initialization. The protocols are highly resilient to churn, i.e., frequent and dynamic topology changes due to addition and deletion of nodes and links. They are also communication efficient because they do not use flooding to discover multi-hop DT neighbors [29].

3.1.4 Adjustment algorithm

During each execution of the adjustment algorithm (see pseudocode in Figure 3.5 with notation defined in Table 3.1), a node *u* may change its position multiple times to find a position in the virtual space with less prediction error. Before algorithm execution, node

u first computes its distances $\tilde{D}(u,v)$ to its physical and DT neighbors using their current virtual positions. (Note that its DT neighbor set and routing costs to DT neighbors do not change during algorithm execution.) Then, u updates its position (executes lines 4-7 of pseudocode) with respect to every multi-hop DT neighbor and some physical neighbors. Specifically, for a physical neighbor v, u updates its position with respect to v if u's distance to v is larger than u's routing cost to v, that is, $\tilde{D}(u,v) > D(u,v)$ (see line 3 of pseudocode). During execution of the adjustment algorithm, there is no message exchange between node u and other nodes. At the end of algorithm execution, node u sends its updated position and position error to all of its physical and DT neighbors.

When node u makes a position adjustment with respect to v, it moves its position in the direction of $[D(u,v) - \tilde{D}(u,v)] \times \hat{u}(x_u - x_v)$, where x_u and x_v are position vectors, and $\hat{u}(x_u - x_v)$ is a unit vector in the direction of $x_u - x_v$. The magnitude of the movement is proportional to the magnitude of $D(u,v) - \tilde{D}(u,v)$, where $D(u,v)$ is routing cost from u to v and $\tilde{D}(u,v)$ is distance between them. If $D(u,v) < \tilde{D}(u,v)$, u moves towards v; if $D(u,v) > \tilde{D}(u,v)$, u moves away from v. The magnitude of the movement is also proportional to the confidence value f of this adjustment computed as follows. If v has a large position error, the position error of v may propagate to u. To mitigate such error propagation, neighbors with large position errors should have less influence in position updates than those with small errors. Similar to Vivaldi, each node u maintains a local variable e_u for its estimated position error. The confidence value f of the adjustment is defined to be

$$f = \frac{e_u}{e_u + e_v}$$

The update rule for each neighbor v that causes a position change is:

$$x_u = x_u + c_c \times f \times [D(u,v) - \tilde{D}(u,v)] \times \hat{u}(x_u - x_v)$$

where c_c is a tuning parameter to be determined (see Section 3.3.4). The value of $D(u,v)$ is available to u in the *cost* field of the tuple in F_u whose *dest* field is v. Note that for a multi-hop DT neighbor v, the cost field does not always store the minimum routing cost from u to v, because the path in the multi-hop DT may not be the shortest one. However, since the main goal of adjusting with a multi-hop DT neighbor v is to move u away from v, we found that an over-estimate of the routing cost works effectively (because if $D(u,v) > \tilde{D}(u,v)$, u moves away from v).

After updating its position, node u also needs to update its estimated position error. For each update caused by neighbor v, u computes the prediction error $\tilde{e}_v$ by

$$\tilde{e}_v = |D(u,v) - \tilde{D}(u,v)| / \tilde{D}(u,v)$$

If v does not cause an update, $\tilde{e}_v = 0$. After checking all neighbors, u computes the average over all of its physical and DT neighbors:

$$e_{new} = \sum \tilde{e}_v / |N_u \cup P_u|$$

The position error of node u is then updated by a moving average:

$$e_u = e_u \times (1 - c_e) + e_{new} \times c_e$$

where c_e is another tuning parameter in the range (0, 1). The initial value of e_u is 1. We use $c_e = 0.25$ in our experiments.

At the end of the adjustment algorithm, node u sends the updated values of x_u and e_u to all neighbors (physical and DT).

3.1.5 Adaptive adjustment timeout

The number of *Adjustment*() executions for node u during an adjustment period is determined by $\lceil \frac{T_a}{\Delta_u} \rceil$, where T_a is the duration of the adjustment period and Δ_u is the adjustment timeout period of node u. One challenge is the choice of a proper value of Δ_u at different stages of the virtual position construction process. At the beginning of an A period, using small timeouts can help nodes rapidly find approximate positions. When node positions are relatively stable, the positions should be refined slowly for them to converge. Also the multi-hop DT constructed in the previous J period needs to be updated after several *Adjustment*() executions. If *Adjustment*() is executed too frequently with an outdated multi-hop DT, node positions may oscillate and do not converge.

We use an adaptive timeout technique to achieve fast and accurate convergence. The initial timeout Δ_{u0} is set to a small value, e.g., 2 sec. After that, each node calculates the average position error of its physical and DT neighbors, denoted by $\bar{e}$. The timeout is then changed to

$$\Delta_u = \min(\Delta_{u0}/\bar{e}, T_a)$$

Note that position errors are initialized to 1 and will decrease with time. When the virtual positions converge and become relatively stable, $\bar{e}$ trends towards 0 and results in a large Δ_u. Experimental results for different values of timeout are presented in Section 3.3.2.

3.2 Greedy Distance Vector (GDV) Routing

In GDV_basic (see pseudocode in the left column of Figure 3.6), when node u has a packet to forward, it uses the virtual positions of its physical neighbors and the destination t to compute estimated routing costs. GDV_basic does not use MDT. Furthermore, GDV_basic does not assume the use of VPoD; virtual positions of nodes may be provided by any virtual positioning protocol that can effectively embed routing costs into a virtual space (like VPoD). For each physical neighbor $y \in P_u$, the estimated routing cost from u to t via y is $R_y = c(u,y) + \tilde{D}(y,t)$. Using GDV_basic, node u selects the physical neighbor v such that $R_v = \min_{y \in P_u} R_y$. To avoid routing loops, GDV requires that $\tilde{D}(v,t) < \tilde{D}(u,t)$, i.e., v is closer to the destination than u in the virtual space. If $R_v < \tilde{D}(u,t)$ is satisfied, then $\tilde{D}(v,t) < \tilde{D}(u,t)$

```
GDV_basic(u, t):
1. For each physical neighbor y,
     $R_y \leftarrow c(u, y) + \tilde{D}(y,t)$;
2. Let $v$ be the physical neighbor
   that minimizes $R_y$;
3. if $R_v < \tilde{D}(u,t)$ then
       send the packet to $v$;
4. else
       GR($u$, $t$);
       // geographic routing
5. end if
```

```
GDV(u, t):
1. For each physical neighbor $y$,
     $R_y \leftarrow c(u, y) + \tilde{D}(y,t)$;
2. For each multi-hop DT neighbor y,
     $R_y \leftarrow D(u, y) + \tilde{D}(y,t)$;
3. Let $v$ be the neighbor that minimizes $R_y$;
4. if $R_v < \tilde{D}(u,t)$ then
       send the packet to $v$ directly or by the
       multi-hop path;
5. else
       MDT_greedy($u$, $t$);
6. end if
```

Figure 3.6: GDV pseudocode at node *u* to destination *t*

holds, and *u* sends the packet to *v*. Note that lines 1-3 in left column of Figure 3.6 perform DV routing with distance vectors computed locally rather than communicated and stored.

In line 4 in left column of Figure 3.6, GDV_basic switches to a geographic routing protocol, GR, based upon greedy forwarding with some recovery method to move packets out of local minima. (Almost any existing geographic routing protocol may be used as GR.) GR uses the virtual positions of nodes for its forwarding decision without any consideration of link costs. When a node, say *w*, receives a packet that is in GR recovery, *w* skips lines 1-3 in the GDV_basic pseudocode and runs GR. (This detail is omitted in Figure 3.6.)

Since a multi-hop DT is already constructed by VPoD, the version of GDV we use in this chapter (see pseudocode in the right column of Figure 3.6) also makes use of the set of multi-hop DT neighbors to improve routing performance and provide guaranteed delivery. We know that MDT forwarding in a correct multi-hop DT provides guaranteed delivery in d-dimensional spaces, $d \geq 2$, as well as a routing stretch close to 1.

Using GDV, when node *u* has a packet to forward, it uses the virtual positions of its physical and multi-hop DT neighbors and the destination *t* to compute estimated routing costs. VPoD provides *u* with the virtual position, routing cost, and a forwarding path to each of its multi-hop DT neighbors. For every multi-hop DT neighbor y, node u computes the estimated routing cost via y to t, $R_y = D(u,y) + \tilde{D}(y,t)$. Using GDV, u selects the node v such that $R_v = \min_{y \in P_u \cup N_u} R_y$. If $R_v < \tilde{D}(u,t)$, u sends the packet to v directly if v is a physical neighbor, or by the multi-hop path to v if v is a multi-hop DT neighbor (line 4 in right column of Figure 3.6).

If $R_v < \tilde{D}(u,t)$ is not satisfied, node u runs MDT-greedy using virtual positions of nodes without any consideration of routing costs (line 5 in right column of Figure 3.6). When a node, say w, receives a packet that is being forwarded in a virtual link and w is not the virtual link's destination, it skips lines 1-4 in the GDV pseudocode and runs MDT-greedy. (This detail is omitted in Figure 3.6.) Since executing line 4 in the GDV pseudocode strictly reduces a packet's distance to its destination in the virtual space, it is straightforward to prove that GDV provides guaranteed delivery because MDT-greedy provides guaranteed delivery.

3.2.1 GDV for different routing metrics

GDV can use any routing metric that DV uses, such as, hop count, latency, ETX, ETT, energy consumption, and propagation distance, etc. Both GDV and DV require a metric m that is positive and additive. The metric, however, may be *asymmetric*, namely, it is not required that $m(u,v) = m(v,u)$ for two physical neighbors, u and v. The following example illustrates GDV's requirement of additivity and non-requirement of symmetry. In MDT protocols, when node a sends a Neighbor_Set_Request message to node b along the path a-x-y-b, the message's routing cost field is initialized to zero at node a. Then node x adds $c(x,a)$ to the field. Later, node y adds $c(y,x)$ to the field. Finally, node b adds $c(b,y)$ to the field. The cumulative value provides node b, the destination of the message, its routing cost back to node a. Subsequently, node b sends a Neighbor_Set_Reply message to a along the reverse path and node a obtains from the message its routing cost to b. Note that the costs of b-a and a-b paths may be different.

When a routing metric captures more network and link characteristics (such as, link quality by ETX [12] and both link quality and capacity by ETT [14]) the metric can be used to provide higher throughput for shortest-path routing. GDV is a geographic routing protocol designed to take advantage of such routing metrics. We found that even when hop count is used as the routing metric, GDV has better routing stretch performance than prior geographic routing protocols. This is because the distance in virtual space is better than the geographic distance in physical space for predicting routing stretch.

3.3 Performance Evaluation

3.3.1 Methodology

We evaluate the performance of GDV using a packet-level discrete-event simulator. Queuing delays are not simulated because we do not evaluate performance metrics that depend on congestion, e.g., end-to-end throughput and latency. Instead, random message delivery times from one node to the next are sampled from a uniform distribution over a specified time interval.

Performance criteria. GDV works for any routing metric that is positive and additive. For this chapter, we used two common metrics in our experiments, namely, hop count and ETX. When using *hop count* as metric, we evaluate the routing stretch of each protocol. The *routing stretch* value between a pair of source and destination nodes is defined to be the ratio of the hop count in the selected route to the hop count in the shortest route in the connectivity graph. When using *ETX* as metric, we evaluate the average *number of transmissions* used to deliver a packet from a source node to a destination node. The routing stretch and number of transmissions shown in the figures are the average values over all source-destination pairs in the network. Using hop count as metric, we compare GDV with MDT-greedy. Using ETX as metric, we compare GDV with NADV [34].[5] To give

[5] PRR$\times$distance [52] is similar to NADV with PRR = 1/ETX.

advantage to NADV and MDT-greedy in the comparisons, we used *accurate node locations* for NADV and MDT-greedy in our experiments.[6]

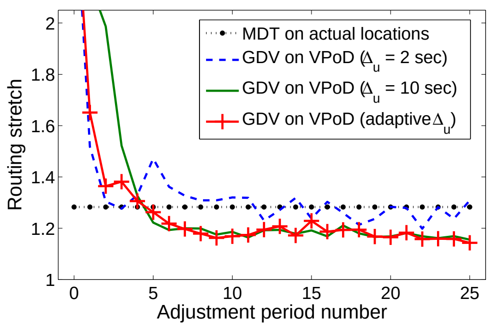
We measure the storage cost of a routing protocol by counting the number of distinct nodes a node needs to know (and store) to perform forwarding, and computing the average value over all nodes. This represents the storage cost of a node's minimum required knowledge of other nodes. It has been validated that the overall storage cost for forwarding is linearly proportional to the number of distinct nodes stored [29]. This metric, unlike counting bytes, requires no implementation assumptions which may cause bias when different routing protocols are compared.

Creating general connectivity graphs and ETX values. We used the link-layer simulator developed by the authors of [52] to create connectivity graphs and link costs (ETX values) of wireless networks. Initially, N nodes are randomly placed in a 2D space. The packet reception rate (PRR) between two nodes is computed as a function of the distance, node density, and other parameters including path loss exponent, shadowing standard deviation, modulation and encoding schemes, output power, noise floor, preamble and frame lengths, and randomness. We use the default values for all parameters [52]. If the packet reception rate between two nodes is greater than 0.1, a physical link is placed between the two nodes in the connectivity graph. The ETX value of the link (in each direction) is the inverse of the PRR value. For wireline networks, we use real ISP topologies by the Rocketfuel project [53] and synthetic topologies generated by BRITE network topology simulator [40].

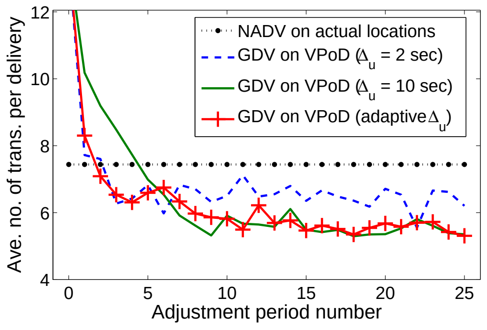
For some experiments, we also randomly placed some large obstacles in the 2D space. Nodes are not placed in space occupied by obstacles. If the line between two nodes intersects any obstacle, there is no physical link between the nodes.

We will first present experimental results for 200-node networks. In section 3.3.7, the number of nodes is varied from 100 to 1000.

[6]MDT-greedy provides guaranteed delivery for nodes with inaccurate or arbitrary coordinates. However, its routing stretch is lower when node locations are known more accurately.



(a) metric is hop count

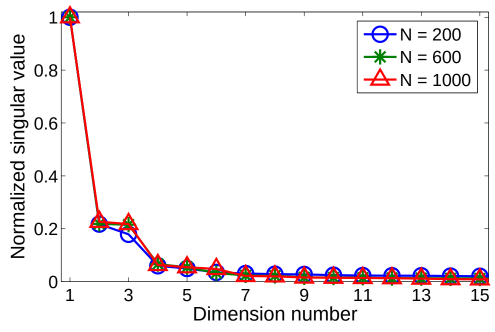


(b) metric is ETX

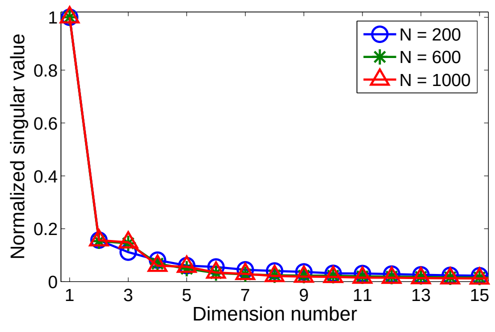
Figure 3.7: Routing performance of GDV, MDT, and NADV for different timeout values

3.3.2 Adaptive adjustment timeout

We conducted many experiments for different values of adjustment timeout. We show representative results for a 200-node network in Figure 3.7. Nodes are in a 100m$\times$100m 2D physical space. The average number of physical neighbors per node is 14.5. VPoD assigns node positions in a 3D virtual space. Routing performance versus adjustment period number (which represents time) is presented for hop count used as metric in Figure 3.7(a) and for ETX used as metric in Figure 3.7(b). The duration of an adjustment period is $T_a = 20$ seconds. Note that when the adjustment timeout is a small value (2 seconds), nodes can find their approximate positions after two periods. However, the routing performance keeps oscillating after that. On the other hand, using a large adjustment timeout (10 seconds) slows down the convergence. Adaptive timeout is the best strategy. Using adaptive timeout, the convergence is as fast as using a small timeout and the quality of virtual positions after convergence is similar to that from using a large timeout. We used adaptive timeout for all other experiments to be presented in this chapter.



(a) metric is hop count



(b) metric is ETX

Figure 3.8: Normalized singular values for different network sizes

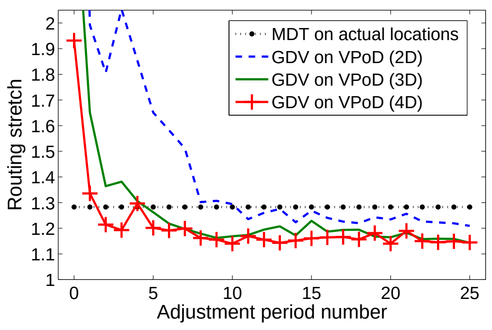
3.3.3 Choice of Dimensionality

We use Principal Component Analysis (PCA) to determine whether a low-dimensional s-pace can be used to effectively model routing costs of practical networks. We then use it to find an appropriate dimensionality to use and we present experimental results to validate the PCA results.

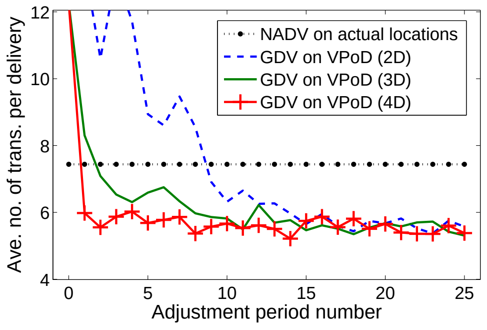
PCA relies on Singular Value Decomposition (SVD). The input of SVD is an $N \times N$ matrix M, where each element m_{ij} is the routing cost from node i to node j. SVD factors M into the product of three matrices: $M = U \cdot S \cdot V^T$, where S is a diagonal matrix with nonnegative elements s_i. The diagonal elements are called *singular values* of M, which are ordered non-increasingly.

From $M = U \cdot S \cdot V^T$, we have $m_{ij} = \sum_{k=1}^{N} s_k u_{ik} v_{jk}$. If singular values $s_1, ..., s_d$ are much larger than the rest, we may approximate m_{ij} by $m_{ij} \approx \sum_{k=1}^{d} s_k u_{ik} v_{jk}$. This means that the routing cost matrix M can be embedded in a d-dimensional Euclidean space with low errors.

Figure 3.8 shows our experimental results for networks of 200, 600 and 1000 nodes. Each data point represents the average result from 20 different networks. The routing costs in the input matrix are measured in hop count for experiments in Figure 3.8(a), and in ETX



(a) metric is hop count



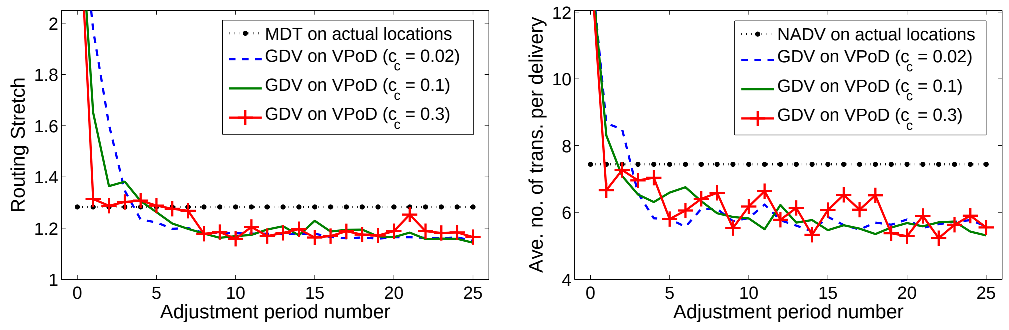
(b) metric is ETX

Figure 3.9: Routing performance of GDV, MDT, and NADV for 2D, 3D, and 4D

for experiments in Figure 3.8(b). The singular values shown are normalized. The first three singular values are much larger than the remaining ones. Also as the network size increases, the third singular value increases in magnitude, which implies that the third dimension is more important for a larger network size.

We have performed many experiments for different networks embedded in 2D, 3D, and 4D virtual spaces. Figure 3.9 shows representative results of routing performance for 2D, 3D and 4D, using the same 200-node network for experiments in Figure 3.7. After 10 adjustment periods, the routing performance of GDV is better than MDT and NADV for all three virtual spaces. For 4D, the routing performance is close to the converged value after just one or two adjustment periods. 2D requires many more adjustment periods to converge. Note that the converged values of 4D are not much better than those of 3D. This observation is consistent with the PCA results in Figure 3.8.

From the PCA and experimental results, 2D or 3D are good choices. As to be shown in Section 3.3.6, both the storage and communication costs of VPoD in 4D are significantly higher than those in 2D or 3D.



(a) metric is hop count (b) metric is ETX

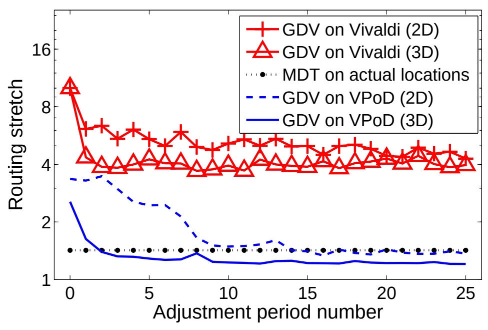
Figure 3.10: Routing performance of GDV, MDT, and NADV for different values of tuning parameter c_c

3.3.4 Impact of tuning parameter

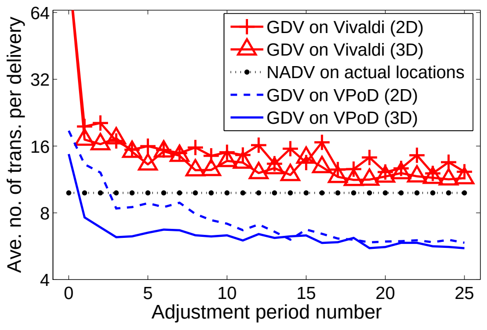
The tuning parameter c_c controls the size of movement in position updates. We tried different values of c_c using the same network used for experiments shown in Figure 3.7. A 3D virtual space is used for VPoD. Figure 3.10 shows that a smaller value ($c_c = 0.02$) causes slower convergence in the first few adjustment periods but its convergence is still quite fast and accurate. When a large value ($c_c = 0.3$) is used, the convergence is fast at the beginning, but there are oscillations in the ETX experiments (see Figure 3.10(b)). VPoD with $c_c = 0.3$ still finds good virtual positions after 20 adjustment periods. Empirically, VPoD is quite robust to different values of c_c because VPoD uses two other adaptive values to control adjustments, i.e., confidence and timeout. We used $c_c = 0.1$ for all other experiments presented in this chapter.

3.3.5 Impact of obstacles

The physical space of practical wireless networks may include large obstacles that block wireless transmissions. Thus we also evaluated GDV for networks with obstacles. In these experiments, each obstacle is a 10m×10m square. We randomly placed four obstacles in a 100m×100m physical space with the same 200-node network used for experiments

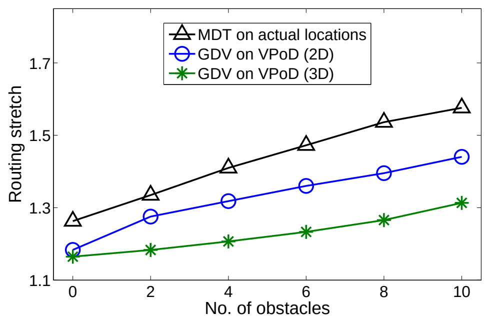


(a) metric is hop count

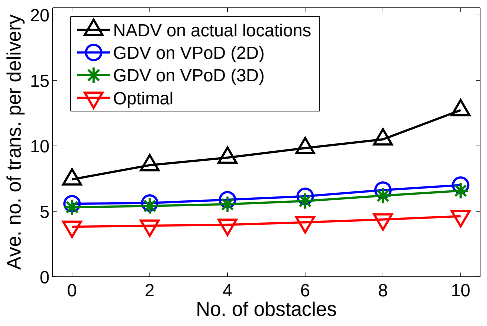


(b) metric is ETX

Figure 3.11: Routing performance of GDV, MDT, and NADV with four randomly placed obstacles



(a) metric is hop count

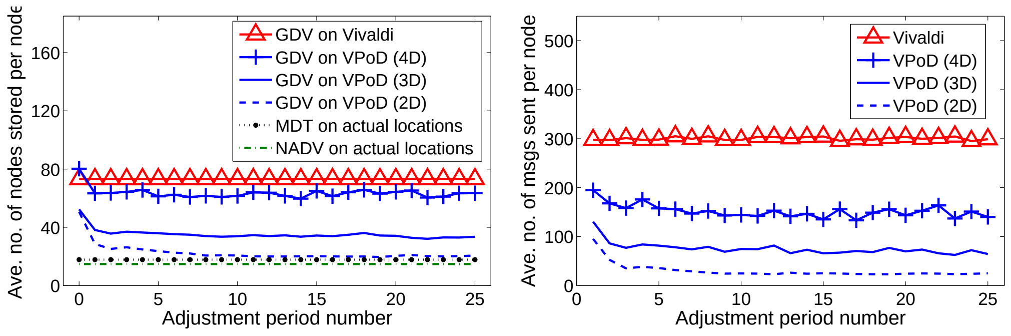


(b) metric is ETX

Figure 3.12: Routing performance of GDV, MDT, and NADV vs. number of obstacles

in Figure 3.7. We also quantitatively compared VPoD with Vivaldi, by running GDV on virtual positions constructed by them. Similar to the experiments in Figure 3.2, we allow each node of Vivaldi to sample both one-hop and two-hop neighbors. The results are shown in Figure 3.11. GDV on VPoD outperforms both MDT and NADV on actual locations and it outperforms GDV on Vivaldi by a very large margin. (We found the performance of GDV on Vivaldi to be consistently poor. For the sake of clarity of presentation, we will omit results for GDV on Vivaldi in Figures 3.12, 3.14, and 3.15 to be presented.)

Next we varied the number of obstacles from 0 to 10 in the 100m×100m physical



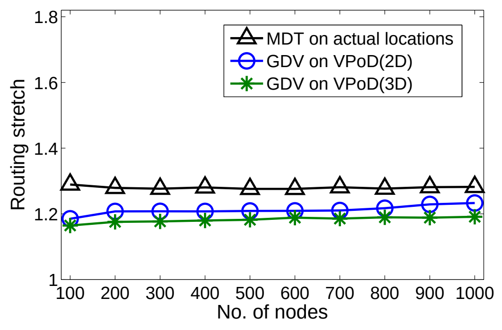
(a) Storage cost (b) control message cost in an adjustment period

Figure 3.13: Storage cost and communication cost of VPoD and Vivaldi

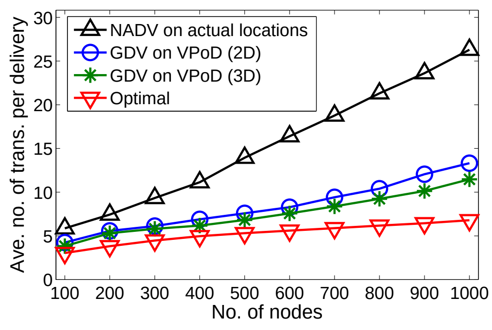
space for 200-node networks. The results are shown in Figure 3.12. Each data point is the average value of 20 simulation runs for 20 different networks. For comparison, we also show the optimal values of shortest path routing using ETX as metric in Figure 3.12(b). In the same figure, the average number of transmissions of NADV increases from 7.44 (0 obstacle) to 12.73 (10 obstacles), while that of GDV on VPoD (3D) increases from 5.31 (0 obstacle) to 6.57 (10 obstacles). Note that the routing performance of GDV on VPoD is fairly close to that of optimal routing.

3.3.6 Storage and communication costs

A multi-hop DT requires extra storage for multi-hop DT neighbors. The amount of extra storage varies during the course of the VPoD construction. At the beginning, most DT neighbors are not physical neighbors because the initial positions are fairly arbitrary. When VPoD has converged, the physical and DT neighbor sets have a large overlap. We evaluated both storage and communication costs using the same 200-node network for experiments in Figure 3.7. Figure 3.13(a) shows storage cost over time. The routing metric is hop count. (Results for the ETX metric are similar and not shown.) All three curves of GDV on VPoD start from high values and then drop after two adjustment periods. The storage cost of VPoD in 2D after convergence is very close to those of MDT and NADV on actual



(a) metric is hop count



(b) metric is ETX

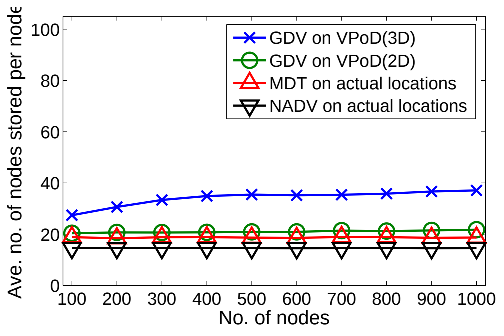
Figure 3.14: Routing performance of GDV, MDT, and NADV versus N

locations. The storage cost of VPoD in 4D after convergence is much higher than those in 2D and 3D, but still lower than that of two-hop Vivaldi. NADV requires each node to store physical neighbors only and has the lowest storage cost.

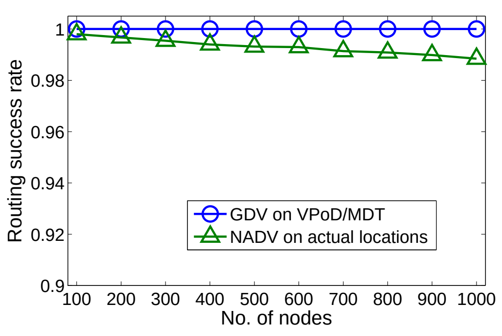
The average number of control messages sent per node in each adjustment period for constructing virtual positions is shown in Figure 3.13(b) for VPoD and Vivaldi. The message cost of VPoD includes both the multi-hop DT construction and adjustment update messages. The routing metric is hop count. (Results for the ETX metric are similar and not shown.) VPoD in 2D has the lowest message cost, which is about 20 messages per join-and-adjustment period. After convergence, the message costs of VPoD in 3D and 4D are about 60 and 140 messages, respectively, per join-and-adjustment period. Two-hop Vivaldi requires many more messages. We do not show message costs for MDT and NADV on actual locations. Given location information, they require a one-time construction with low message costs. But they require localization methods which have message and other costs to provide accurate location information.

3.3.7 Varying the number of nodes

We evaluate the performance of GDV for network size (N) from 100 to 1000 nodes. For 200-node experiments, the size of the physical space is 100m$\times$100m. For a smaller (or



(a) Storage cost



(b) Routing success rate

Figure 3.15: Storage cost and routing success rate of GDV, MDT, and NADV versus N

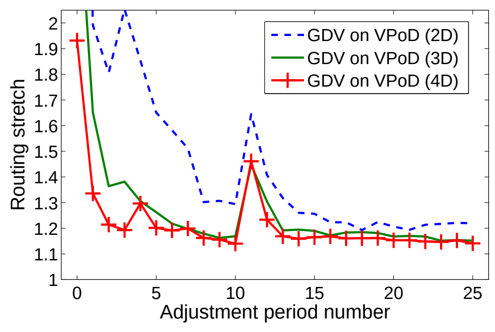
larger) number of nodes, the size of the physical space is scaled down (or up) proportionally such that the average number of physical neighbors per node is kept at 14.5. No obstacles are placed. Each data point shown is the average value of 20 simulation runs for 20 different networks.

Figure 3.14(a) shows routing stretch versus N. GDV on VPoD performs better than MDT on actual locations. (Note that MDT has been shown to provide the lowest routing stretch when compared to other geographic routing protocols [29].) The routing stretch values of both GDV and MDT remain low as N increases.

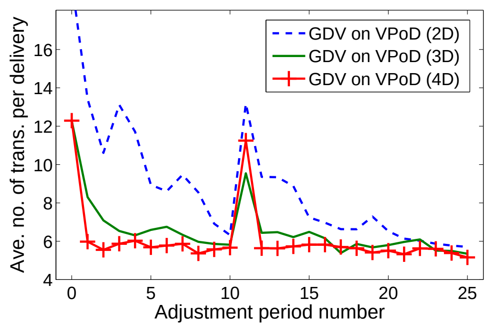
Figure 3.14(b) shows that the average number of transmissions increases with N for all protocols (including optimal routing). NADV increases a lot more than GDV. For $N=$1000, the average number of transmissions of GDV is only half of that of NADV.

Figure 3.15(a) shows storage cost versus N. NADV has the lowest cost, followed in order by MDT, GDV on VPoD (2D), and GDV on VPoD (3D). The storage costs for all protocols remain low as N increases.

Figure 3.15(b) shows the routing success rates of different protocols. GDV and MDT both provide guaranteed delivery (the routing success rate was 100% in every experiment). The routing success rate of NADV is below 100% and decreases with N because NADV's recovery method from local minima does not work well for general connectivity



(a) metric is hop count



(b) metric is ETX

Figure 3.16: Routing performance of GDV under churn

graphs used in the experiments.

3.3.8 Resilience to dynamic topology changes

GDV and VPoD are highly resilient to dynamic network topology changes (*churn*) because VPoD uses MDT which is highly resilient. For the same 200-node network used for the experiments in Figure 3.9, we introduced node churn after the 10th adjustment period and at the beginning of the 11th join period. At the same time, 150 nodes (out of 200 nodes) failed and 150 new nodes joined. Each failed node became silent. Each new node chose its position in the virtual space to be the center of the positions of its physical neighbors that have a position error less than 1. Its own position error is set to 1.

Figure 3.16 shows the routing performance of GDV using VPoD in 2D, 3D, and 4D. Note that the routing performance for each routing metric (hop count or ETX) becomes worse immediately after churn. However, routing performance quickly converges to a low value after several adjustment periods (just 2-3 periods for 3D). The routing performance after 20 periods in total is as good as the performance shown in Figure 3.9 for experiments with a static topology. These and similar results from other churn experiments show that GDV and VPoD are very resilient to dynamic topology changes.

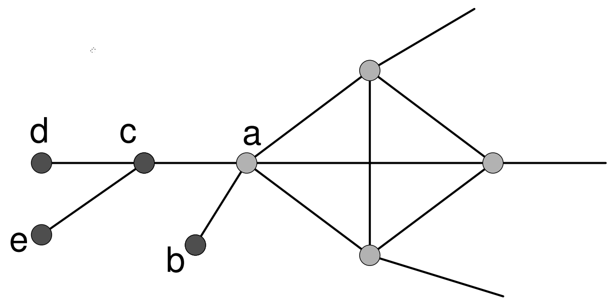


Figure 3.17: An example of pruning

3.4 Performance of GDV on wireline networks

GDV has minimal assumptions: connected graph and bidirectional links. Hence GDV can also be applied on wireline networks for intra-domain or layer-2 routing. In this section we evaluate the performance of GDV using real and synthetic wireline network topologies.

We first use three router-level topologies collected by the Rocketfuel project [53], each of which belongs to a single AS. We discover that a real router-level topology usually contains some nodes that have only one or two neighbors, e.g., some edge routers. Maintaining a multi-hop DT for a network including these nodes is very inefficient. It is because most DT neighbors of these nodes are multi-hop neighbors, incurring both high communication and storage cost. Furthermore, these nodes do not have enough physical neighbors for position adjustment. To deal with this problem, we design two schemes, *pruning* and *two-hop neighbor support*, to improve the performance of GDV on real router-level topologies.

Pruning: A node with one physical neighbor does not need to participate in the multi-hop DT, because it has only one access to other nodes, such as nodes *b*, *d* and *e* in the example of Figure 3.17. Before starting VPoD, each of these nodes sends a *PRUNED* message to its *parent*, i.e., the only physical neighbor. The *PRUNED* message indicates that the sender declines to participate in the multi-hop DT, and will simply use the parent's position as its own position for receiving data messages. When a node, say *c* in Figure 3.17, finds that there is only one physical neighbor that is not pruned (node a), it also sends a *PRUNED* message to the physical neighbor. Hence nodes b, c, d and e all use a's position, and a will store their identifiers to forward messages to them. In this way, any tree sub-graph attached to the remaining network topology will be pruned, and only the root participates in the multi-hop DT. Every node in the tree uses the root's virtual position. A data message whose destination is in the tree will be routed to the root and then forwarded.

Two-hop neighbor support: For nodes that do not have enough physical neighbors for position adjustment, they can randomly explore some two-hop neighbors and include them into their physical neighbor sets. For each two-hop neighbor w stored by node u, the "link cost" $c(u,w)$ is assigned to be $\min_{v \in P_u \bigcap P_w} c(u,v) + c(v,w)$. The *succ* field in the corresponding tuple is the physical neighbor that minimizes $c(u,v) + c(v,w)$. Since GDV and VPoD use any additive routing metric, a two-hop neighbor can be considered the same as a physical neighbor in protocol execution.

To determine whether a low-dimensional space can represent accurately the routing cost of router-level networks, we show in Figure 3.18 the normalized singular values of the routing cost matrices, after pruning, of AS1755, AS3967 and AS6461. The first two or three singular values are much larger than the remaining ones for all three topologies. From the results, VPoD in low dimensions (2D-4D) is good for real router-level topologies.

Figure 3.19 shows the GDV routing stretch on the AS1755 network in 2D and 3D, with and without pruning and two-hop neighbor support. The original GDV on VPoD has relatively high routing stretch. The converged routing stretch values are about 2.5 and 1.8 for 2D and 3D, respectively. The reason might be that some nodes with few physical neighbors are not able to determine their proper positions. However, pruning and two-hop neighbor support significantly improve the routing stretch. After applying these schemes, the converged values are 1.27 and 1.15 for 2D and 3D respectively. Experiments on AS3967 and AS6461 topologies show similar results.

We then conduct experiments on synthetic router-level topologies generated by the

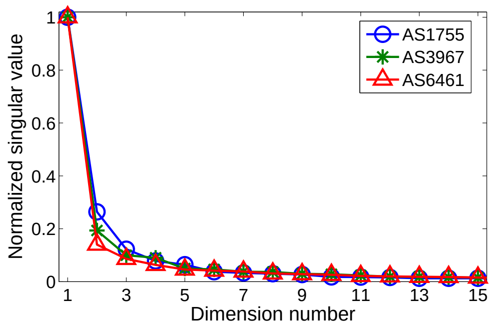


Figure 3.18: Normalized singular values for Rocketfuel router-level topologies

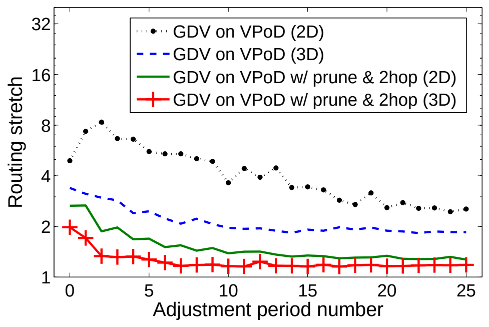
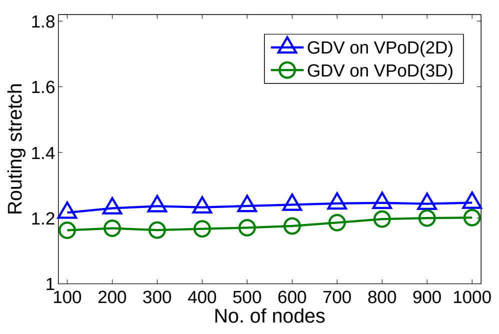
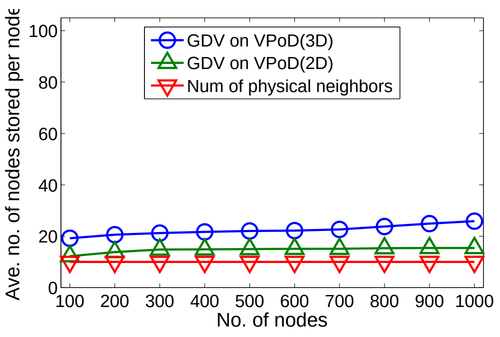


Figure 3.19: GDV Routing stretch on AS1755 topology

BRITE internet topology generator [40] to evaluate the performance of GDV for networks with different values of N. Each data point shown is the average value of 10 simulation runs for 10 different networks. Figure 3.20(a) shows routing stretch versus N, and Figure 3.20(b) shows storage cost versus N. Similar to the results of GDV on wireless topologies, the routing stretch and storage cost remain low as N increases, for both 2D and 3D.



(a) Routing stretch



(b) Storage cost

Figure 3.20: Routing stretch and storage cost of GDV versus *N*

Chapter 4

ROME Protocols

4.1 Unicast and Multicast Routing in ROME

Consider a network of switches with an arbitrary topology (any connected graph). Each switch selects one of its MAC addresses to be its identifier. End hosts are connected to switches which provide frame delivery between hosts. Ethernet frames for delivery are encapsulated in ROME packets. Switches interact with hosts by Ethernet frames using conventional Ethernet format and semantics. ROME protocols run only in switches. Link-level delivery is assumed to be reliable.

4.1.1 Virtual space for switches

A Euclidean space (2D, 3D, or a higher dimension) is chosen as the virtual space. The number of dimensions and the minimum and maximum coordinate values of each dimension are known to all switches. Each switch determines for itself a location in the space represented by a set of coordinates.

Location hashing. To start ROME protocols, each switch boots up and assigns itself an initial location randomly by hashing its identifier, ID_S, using a globally-known hash function H. The hash value is a binary number which is converted to a set of coordinates.

Our protocol implementation uses the hash function MD5 [50], which outputs a 16-byte binary value. 4 bytes are used for each dimension. Thus locations can be in 2D, 3D, or 4D.[1]

Consider, for example, a network that uses a 2D virtual space. For 2D, the last 8 bytes of $H(ID_S)$ are converted to two 4-byte binary numbers, x and y. Let MAX be the maximum 4-byte binary value, that is, $2^{32}-1$. Also let min_k and max_k be the minimum and maximum coordinate values for the kth dimension. Then the location in 2D obtained from the hash value is $(min_1 + \frac{x}{MAX}(max_1 - min_1),\ min_2 + \frac{y}{MAX}(max_2 - min_2))$, where each coordinate is a real number. The location can be stored in decimal format, using 4 bytes per dimension. Hereafter, for any identifier, ID, we will use $H(ID)$ to represent its location in the virtual space and refer to $H(ID)$ as the identifier's *location hash* or, simply, *location*.

Switches discover their directly-connected neighbors and, using their initial locations, proceed to construct a multi-hop DT [29]. Switches then update their locations using VPoD and construct a new multi-hop DT as described in Subsection 3.1.

Unicast routing. Unicast packet delivery in ROME is provided by GDV routing in the multi-hop DT maintained by switches. In a correct multi-hop DT, *GDV routing provides guaranteed delivery of any packet to the switch that is closest to the packet's destination location* [29, 46].

4.1.2 Hosts

Hosts have IP and MAC addresses. Each host is directly connected to a switch called its *access switch*. An access switch knows the IP and MAC addresses of every host connected to it. The routable address of each host is the location of its access switch in the virtual space, also called the *host's location*. Hosts are not aware of ROME protocols and run ARP [44], DHCP [15], and Ethernet protocols in the same way as when they are connected to a conventional Ethernet.

[1]Conceptually, a higher dimensional space gives VPoD more flexibility but requires more storage space and control overhead. Our experimental results show that VPoD's performance in 2D is already very good.

4.1.3 Stateless multicast and its applications

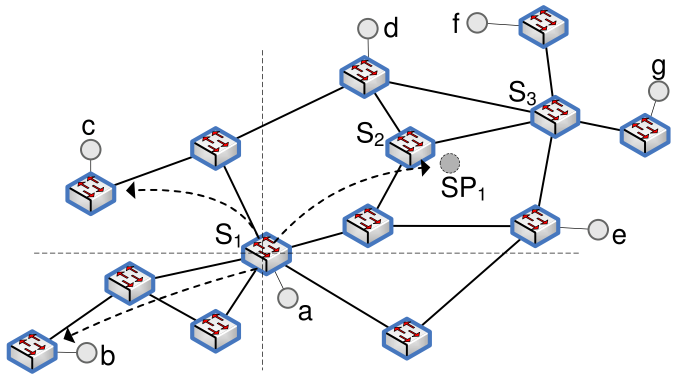
To provide the same services as conventional Ethernet, ROME needs to support group-wide broadcast or multicast, for applications, such as, VLAN, teleconferencing, television, replicated storage/update in data centers, etc.

A straightforward way to deliver messages to a group is by using a multicast tree similar to IP multicast [25]. When there are many groups with many hosts in each group, the amount of multicast state stored in switches can become a scalability problem.

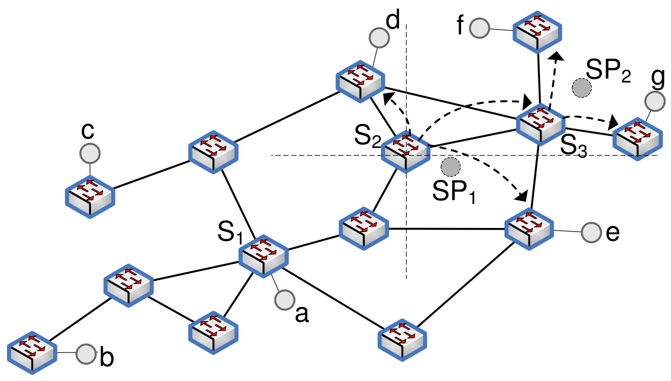
We present a *stateless multicast* protocol for group-wide broadcast in ROME. A group message is delivered using the locations of its receivers without construction of any multicast tree. Switches do not store any state for delivering group messages.

The membership information of stateless multicast is maintained at a *rendezvous point* (RP) for each group. The RP of a group is determined by the location hash $H(ID_G)$, where ID_G is the group's ID. The switch whose location is closest to $H(ID_G)$ serves as the group's RP. The access switch of the sender of a group message sends the message to the RP by unicast. GDV routing guarantees message delivery to the switch closest to $H(ID_G)$.

The RP then forwards the message to other group members (receivers) as follows: The RP partitions the entire virtual space into multiple regions. To each region with one or more receivers, the RP sends a copy of the group message with the region's receivers (their locations) in the message header (actually the ROME packet header). The destination of the group message for each region is a location, called *split position* (SP), which is either (i) the closest receiver location in that region, or (ii) the mid-point of the two closest receiver locations in the region. By GDV routing, the group message will be routed to a switch closest to the SP. This switch will in turn partition its region into multiple sub-regions and send a copy of the group message to the SP of each sub-region. Thus a multicast tree rooted at the RP grows recursively until it reaches all receivers. The tree structure is not stored anywhere. At each step of the tree growth, a switch computes SPs for the next step based on receiver locations in the group message it is to forward.



(a) Split at S_1



(b) Splits at S_2 and S_3

Figure 4.1: Example of stateless multicast

We present an example of stateless multicast in Figure 4.1(a). The group consists of 7 hosts a,b,c,d,e,f,g, connected to different switches with locations in a 2D virtual space as shown. Switch S_1 serves as the RP. Host a sends a message to the group by first sending it to S_1. Upon receiving the message, S_1 realizes that it is the RP. S_1 partitions the entire virtual space into four quadrants and sends a copy of the message by unicast to each of the 3 quadrants with at least one receiver. The message to the northeast quadrant with four receivers (d,e,f, and g) is sent to a split position, SP_1, which is the midpoint between the locations of d and e, the two receivers closest to S_1. The message will then be routed by

GDV to S_2, the switch closest to SP_1.

Subsequently, S_2 partitions the space into four quadrants and sends a copy of the message to each of the three quadrants with one or more receivers (see Figure 4.1(b)). For the northeast quadrant that has two receivers, the message is sent to the split position, SP_2, which is the midpoint between the locations of f and g. The message to SP_2 will be routed by GDV to S_3, the switch closest to SP_2, which will unicast copies of the message to f and g.

At any time during the multicast, when a switch realizes that a receiver is a directly-connected host, it can transmit the message directly to the host and removes the host from the set of receivers in the message to be forwarded.

In ROME, for each group, its group membership information is stored in only one switch, the group's RP. For this group, no multicast state is stored in any other switch. This is a major step towards scalability. The tradeoff for this gain is an increase in communication overhead from storing a set of receivers in the ROME header of each group message. Experimental results in Subsection 5.6 show that this communication overhead is small. This is because when the group message is forwarded by the RP and other switches, the receiver set is partitioned into smaller and smaller subsets.

The implementation of stateless multicast, as described, is not limited to the use of a 2D space. Also, partitioning of a 2D space at the RP, or at a switch closest to a SP, is not limited to four quadrants. The virtual space can be partitioned into any number of regions evenly or unevenly. A study of other virtual spaces and partitioning methods for implementing stateless multicast will be future work.

Stateless multicast for VLAN. Members of a VLAN are in a logical broadcast domain; their locations may be widely distributed in a large-scale Ethernet. ROME's stateless multicast protocol is used to support VLAN broadcast. When a switch detects that one of its hosts belongs to a VLAN, it sends a Join message to location $H(ID_V)$, where ID_V is the VLAN ID. By GDV, The Join message is routed to the switch closest to $H(ID_V)$, which is the RP of the VLAN. The RP then adds the host to the VLAN membership. The protocol for a host to leave a VLAN is similar. VLAN protocols in ROME are much more efficient than the current VLAN Trunking Protocol used in conventional Ethernet [2]. The number of global VLANs is restricted to 4094 in conventional Ethernet [21]. There is no such restriction in ROME because stateless multicast does not require switches to store VLAN information to perform forwarding.

4.2 Host and Service Discovery in ROME

Suppose a host knows the IP address of a destination host from some upper-layer service. To route a packet from its source host to its destination host, switches need to know the MAC address of the destination host as well as its location, i.e., location of its access switch. Such address and location resolution are together referred to as *host discovery*.

4.2.1 Delaunay distributed hash table

The benefits of using a DHT for host discovery include the following: (i) uniformly distributing the storage cost of host information over all network switches, and (ii) enabling information retrieval by unicast rather than flooding. The one-hop DHT in SEATTLE [25] uses *consistent hashing* of identifiers into a circular location space and requires that every switch knows *all* other switches. Such global knowledge is made possible by link-state *broadcast*, which limits scalability.

In ROME, the Delaunay DHT (or D^{2}HT) uses *location hashing* of identifiers into a Euclidean space (2D, 3D, or a higher dimension) as described in Subsection 4.1.1. D^{2}HT uses greedy routing (GDV) in a multi-hop DT where every switch only needs to know its directly-connected neighbors and its neighbors in the DT graph. Furthermore, each switch uses a very efficient search method to find its multi-hop DT neighbors without broadcast [29].

In D^{2}HT, information about host i is stored as a key-value tuple, $t_i =< k_i,\ v_i >$,

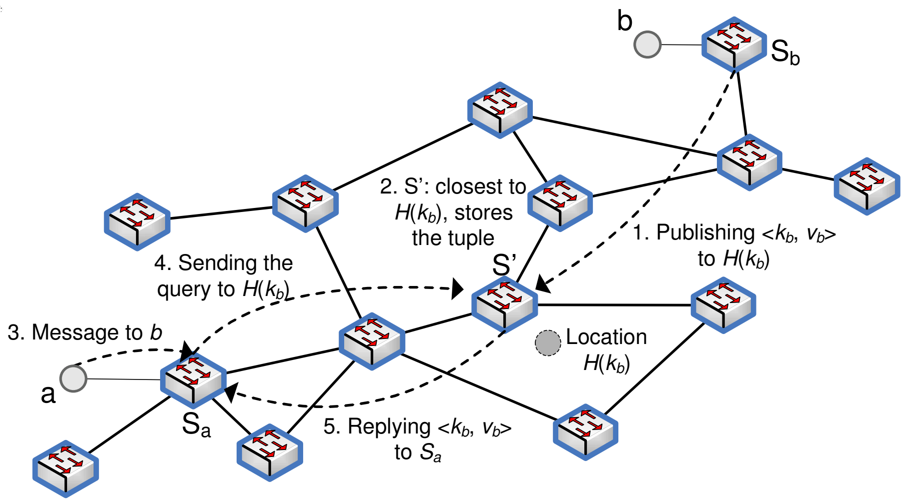

b
S_b
2. S': closest to H(k_b), stores the tuple
S'
1. Publishing <k_b, v_b> to H(k_b)
4. Sending the query to H(k_b)
3. Message to b
Location H(k_b)
a
S_a
5. Replying <k_b, v_b> to S_a

Figure 4.2: S_b publishes a tuple of b. S_a performs a lookup of b

where the key k_i may be the IP (or MAC) address of i, and v_i is host information, such as its MAC address, location, etc. The access switch of host i is the *publisher* of i's tuples. A switch that stores $< k_i, v_i >$ is called a *resolver* of key k_i. The tuples are stored as *soft state*.

To publish a tuple, $t_i =< k_i, v_i >$, the publisher computes its location $H(k_i)$ and sends a publish message of t_i to $H(k_i)$. Location hashes are randomly distributed over the entire virtual space. It is possible but unlikely that a switch exists at the exact location $H(k_i)$. The publish message is routed by GDV to the switch whose location is closest to $H(k_i)$, which then becomes a resolver of k_i. When some other switch needs host i's information, it sends a lookup request message to location $H(k_i)$. The lookup message is routed by GDV to the resolver of k_i, which sends the tuple $< k_i, v_i >$ to the requester. A publish-lookup example is illustrated in Figure 4.2.

4.2.2 Host discovery using D^{2}HT

In ROME, the routable address of host i is i's location c_i, which is the location of its access switch. There are two key-value tuples for each host, for its IP-to-MAC and MAC-to-location mappings.

In a tuple for host i, the key k_i may be its IP or MAC address. If k_i is the MAC address, value v_i includes location c_i and the unique ID, S_i, of i's access switch. If k_i is the IP address, the value v_i includes the MAC address, MAC_i, as well as c_i and S_i. Note that the host location is included in both tuples for each host.

After a host i is plugged into its access switch S_i with location c_i, the switch learns the host's IP and MAC addresses, IP_i and MAC_i, respectively [25]. S_i then constructs two tuples: $< MAC_i, c_i, S_i >$ and $< IP_i, MAC_i, c_i, S_i >$, and stores them in local memory. S_i then sends publish messages of the two tuples to $H(IP_i)$ and $H(MAC_i)$.

Note that each switch stores two kinds of tuples. For a tuple with key k_i stored by switch S, if S is i's access switch, the tuple is a *local tuple* of S. Otherwise, the tuple is published by another switch and is an *external tuple* of S. Switches store key-value tuples as *soft state*.

Each switch interacts with directly-connected hosts using frames with conventional Ethernet format and semantics. When a host j sends its access switch S_j an ARP query frame with destination IP address IP_i and the broadcast MAC address, S_j sends a lookup request to location $H(IP_i)$, which is routed by GDV to a resolver of IP_i. The resolver sends back to S_j the tuple $< IP_i, MAC_i, c_i, S_i >$. After receiving the tuple, the access switch S_j caches the tuple and transmits a conventional ARP reply frame to host j. When j sends an Ethernet frame with destination MAC_i, the access switch S_j retrieves location c_i from its local memory and sends the Ethernet frame to c_i. If S_j cannot find the location of MAC_i in its local memory because, for instance, the cached tuple has been overwritten, it sends a lookup request which is routed by GDV to $H(MAC_i)$ to get the MAC-to-location mapping of host i.

All publish and lookup messages are unicast messages. Host discovery in ROME is accomplished *on demand* and is *flooding-free*.

4.2.3 Reducing lookup latency

Reducing the lookup latency for host discovery is a significant concern. We designed and evaluated three techniques to speed up key-value lookup, including using multiple independent hash functions to publish each key-value tuple at multiple locations, hashing to a smaller region in the virtual space, and taking shortcuts for fast responses.

Multiple location hashes. For each host i, the access switch S_i can publish more than one copy of the tuple $< k_i,\ v_i >$ into the network: S_i applies m independent hash functions on k_i and gets m different location hashes $H_1(k_i)$, ..., $H_m(k_i)$. It then sends m publish messages to these locations and let switches closest to them store the key-value mapping. If a switch wants to request $< k_i,\ v_i >$, it first computes the m location hashes, evaluates the distances from its own location to the m locations, and then selects the nearest one as the destination of the lookup request. According to the property of the VPoD virtual space, shorter distance means approximately lower routing cost or latency. Thus using more hash functions trades a higher storage cost for a lower latency. We will demonstrate this trade-off in the experimental results.

Hashing to a smaller region: To further reduce the lookup latency in ROME, switches can map location hashes to a smaller virtual region. For example, suppose the 2D virtual space of switches is ([0, 100], [0, 100]), and virtual distance is an accurate estimate of routing latency. If location hashes are distributed over the entire virtual space, the worst-case latency between sending a lookup request and receiving its reply is 282.8. If the hash results are mirrored to a smaller region ([25, 75], [25, 75]), the worst-case lookup latency is 212.1. The average latency is also reduced by using a smaller hash region. However, this technique can introduce load imbalance among switches: switches in the region have to store more mapping tuples than the switches outside.

Shortcuts. If there exist a small number of popular hosts in the network, caching is an effective way to provide fast response to lookup request. Each access switch can maintain a cache list that stores the locations of the most popular hosts requested by the hosts attached to it.

A lookup request sent to location $H(k_i)$ does not have to reach the resolver that is closest to $H(k_i)$. Any intermediate switch S can immediately reply to the request for host information and stop forwarding it, under one of three conditions: (1) S is the access switch of i; (2) S is a resolver of i for a different hash function; (3) i's location was previously discovered by S and stored in S's cache.

4.2.4 Maintaining consistent key-value tuples

A key-value tuple $< k_i, v_i >$ stored as an external tuple in a switch is *consistent* iff (i) the switch is closest to the location $H(k_i)$ among all switches in the virtual space, and (ii) c_i is the correct location of i's access switch. At any time, some key-value tuples may become inconsistent as a result of host or network dynamics.

Host dynamics. A host may change its IP or MAC address, or both. A host may change its access switch, such as, when a mobile node moves to a new physical location or a virtual machine migrates to a new system.

Network dynamics. These include the addition of new switches or links to the network as well as deletion/failure of existing switches and links. MDT and VPoD protocols have been shown to be highly resilient to network dynamics (churn) [29, 46]. Switch states of the multi-hop DT as well as switch locations in the virtual space recover quickly to correct values after churn. The following discussion is limited to how host and network dynamics are handled by switches in the role of publisher and in the role of resolver in D^{2}HT.

As a publisher, each switch ensures that local tuples of its hosts are correct when there are host dynamics. For example, if a host has changed its IP or MAC address, the host's tuples are updated accordingly. If a new host is plugged into the switch, it creates tuples for the new host. New as well as updated tuples are published to the network. In addition to these reactions to host dynamics, switches also periodically refresh tuples they previously published. For every local tuple $< k_i,\ v_i >$, S sends a refresh message every T_r second to its location $H(k_i)$. The purpose of a refresh message is twofold: (i) If the switch closest to location $H(k_i)$ is the current resolver, timer of the soft-state tuple in the resolver is refreshed. (ii) If the switch closest to $H(k_i)$ is different from the current resolver, the refresh message notifies the switch to become a resolver.

As a resolver, each switch sets a timer for every external tuple stored in local memory. The timer is reset by a request or refresh message for the tuple. If a timer has not been reset for T_e time, timeout occurs and the tuple will be deleted by the resolver. T_e is set to a value several times that of T_r.

For faster recovery from network dynamics, we designed and implemented a technique, called *external tuple handoff*. When a switch detects topology or location changes in the multi-hop DT, it checks the location $H(k_i)$ of every external tuple $< k_i,\ v_i >$. If the switch finds a physical or DT neighbor closer to $H(k_i)$ than itself, it sends a *handoff message* including the tuple to the closer neighbor. The handoff message will be forwarded by GDV until it reaches the switch closest to $H(k_i)$, which then becomes the tuple's new resolver.

4.2.5 Using Multicast to publish/update tuples

A switch may need to send publish or refresh messages for multiple tuples at a same time. For example, a switch can use the same timer to control the time T_r to send refresh messages for all tuples. In these cases, switches may use the stateless multicast protocol discussed in Section 4.1.3 to send the publish or refresh messages. Experimental results in 5.6 show that using multicast can greatly reduce the control overhead compared to using unicast.

4.2.6 DHCP server discovery using D^{2}HT

In a conventional Ethernet, a new host broadcasts a Dynamic Host Configuration Protocol (DHCP) discover message to find a DHCP server. Each DHCP server that has received the discover message allocates an IP address and broadcasts a DHCP offer message to the host. The host broadcasts a DHCP request to accept an offer. The selected server broadcasts a DHCP ACK message. Other DHCP servers, if any, withdraw their offers.

In ROME, the access switch of each DHCP server publishes the server's information to a location using a key known by all switches, such as, "DHCPSERVER1". When the access switch of a host receives a DHCP discover message, the message is routed by GDV to the location of a DHCP server, without use of flooding. There is no duplicate DHCP offer. To be compatible with a conventional Ethernet, the access switch replies to the host with a DHCP offer and later transmits a DHCP ACK in response to the host's DHCP request.

4.3 ROME for Hierarchical Ethernet

A metropolitan or wide area Ethernet spanning across a large geographic area typically has a hierarchical structure comprising many *access networks* interconnected by a *core network* [22]. Each access network has one or more *border switches*. The border switches of all access networks form the core network. Consider a hierarchical network consisting of 500 access networks each of which has 2000 switches. The total number of switches is 1 million. At 100 hosts per switch, the total number of hosts is 100 millions. We believe that a 2-level hierarchy is adequate for metropolitan scale in the foreseeable future.

4.3.1 Routing in a hierarchical network

For hierarchical routing in ROME, separate virtual spaces are specified for the core network and each of the access networks, called *regions*. Every switch knows the virtual space of its region (i.e., dimensionality as well as maximum and minimum coordinate values of each dimension). Every border switch knows two virtual spaces, the virtual space of its region and the virtual space of the core network, called *backbone*.

The switches in a region first discover their directly-connected neighbors. They then use MDT and VPoD protocols to determine their locations in the region's virtual space

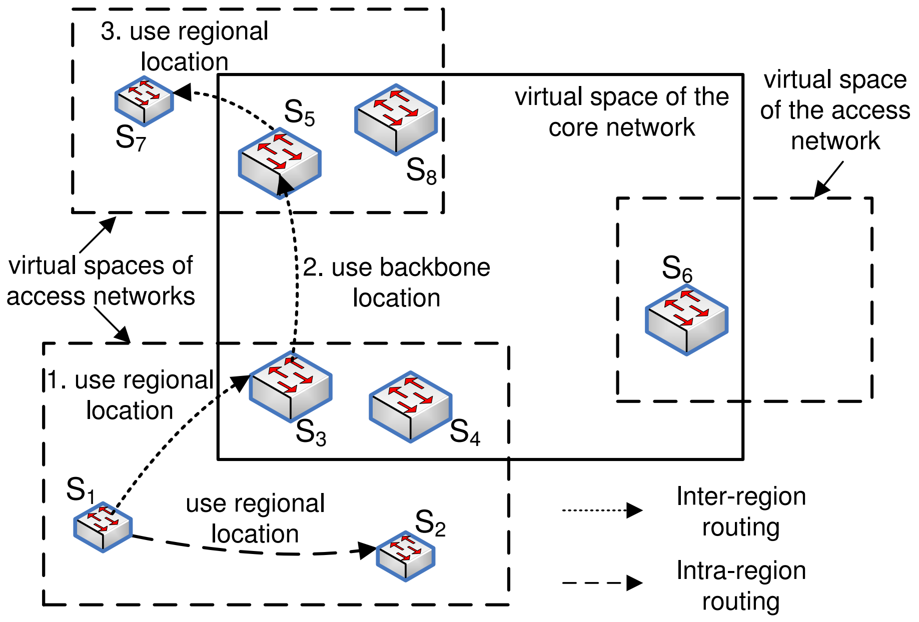


Figure 4.3: Routing in a hierarchical Ethernet

(*regional locations*) and construct a multi-hop DT for the access network. Similarly, the border switches use MDT and VPoD protocols to determine their locations in the virtual space of the backbone (*backbone locations*) and construct a multi-hop DT for the core network. Each border switch sends its information (unique ID, regional and backbone locations) to all switches in its region.

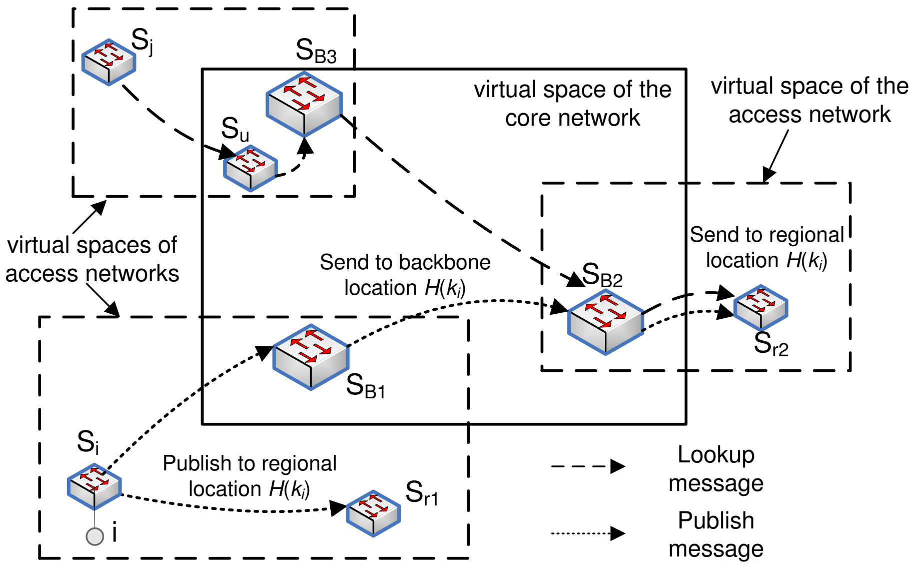
The Delaunay DHT requires the following extension for hierarchical routing: Each key-value tuple $< k_i, v_i >$ of host i stored at a resolver includes additional information, B_i, which specifies the IDs and backbone locations of the border switches in host i's region.

When a host sends an Ethernet frame to another host, its access switch obtains, from its cache or using host discovery, the destination host's key-value tuple, which includes border switch information of the destination region. This information allows the access switch to determine whether to route the frame to its destination using *intra-region* routing or *inter-region* routing.

Intra-region routing. The sender's access switch indicates in the ROME packet header that this is an intra-region packet. The routable address is the regional location of the access switch of the receiver. The packet will be routed by GDV to the access switch of the receiver as previously described. In the example of Figure 4.3, an intra-region packet is routed by GDV from access switch S_1 to destination host's access switch S_2 in the same regional virtual space.

Inter-region routing. For a destination host in a different region, an access switch learns, from the host's key-value tuple, information about the host's border switches and their backbone locations. This information is included in the ROME header encapsulating every Ethernet frame destined for that host. We describe inter-region routing of a ROME packet as illustrated in Figure 4.3. The origin switch S_1 computes its distances in the regional virtual space to the region's border switches, S_3 and S_4. S_1 chooses S_3 which is closer to S_1 than S_4. The packet is routed by GDV to S_3 in the regional virtual space. S_3 learns from the ROME packet header, S_5 and S_8, border switches in the destination's region. S_3 computes their distances to destination S_7 in the destination region's virtual space. S_3 chooses S_5 because it is closer to the destination location. The packet is then routed by GDV in the backbone virtual space to S_5. Lastly, the packet is routed, in the destination region's virtual space, by GDV from S_5 to S_7, which extracts the Ethernet frame from the ROME packet and transmits the frame to the destination host.

Note that at the border switch S_3, it has a *choice of minimizing the distance traveled by the ROME packet in the backbone virtual space or in the destination region's virtual space*. In our current ROME implementation, the distance in the destination region's virtual space is minimized. This is based upon our current assumption that the number of switches in an access network is larger the number of switches in the core network. This choice at a border switch is programmable and can be easily reversed. Lastly, it is not advisable to use the sum of distances in two different virtual spaces (specified independently) to determine routing because they are not comparable. This restriction may be relaxed but it is beyond



S_j
S_{B3}
S_u
virtual space of the core network
virtual space of the access network
virtual spaces of access networks
Send to backbone location $H(k_i)$
S_{B2}
Send to regional location $H(k_i)$
S_{r2}
S_{B1}
S_i
Publish to regional location $H(k_i)$
S_{r1}
i
Lookup message
Publish message

Figure 4.4: Tuple publishing and lookup in a hierarchical Ethernet

the scope of this dissertation.

4.3.2 Host discovery in a hierarchical network

As illustrated in Figure 4.4, the key-value tuple $< k_i, v_i >$ of host i is published to two resolvers in the entire network, namely: a *regional resolver* and a *global resolver*. The regional resolver is the switch closest to location $H(k_i)$ in the same region as host i; it is labeled by S_{r1} in the figure. The publish and lookup protocols are the same as the ones presented in Subsection 4.2.2. To find a tuple with key k_i, a switch sends a lookup message to position $H(k_i)$ in its own region. A regional resolver provides fast responses to queries needed for intra-region communications.

However, switches outside of the host's region cannot find its regional resolver. Therefore, the key-value tuple $< k_i, v_i >$ of host i is also stored in a global resolver to respond to host discovery for inter-region communications. The global resolver can be found by any switch in the entire network. As shown in Figure 4.4, to publish a tuple

$< k_i, v_i >$ to its global resolver, the publish message is first routed by GDV to the regional location of one of the border switches in the region, labeled by S_{B1} in the figure. S_{B1} computes location $H(k_i)$ in the backbone virtual space and includes it with the publish message which is routed by GDV to the border switch closest to backbone location $H(k_i)$ in the core network, labeled by S_{B2} in the figure. Switch S_{B2} serves as the global resolver of host i if it has enough memory space. Switch S_{B2} can optionally send the tuple to a switch in its region such that all switches in the region share the storage cost of the global resolver function (called *two-level location hashing*). In two-level location hashing, the publish message of tuple $< k_i, v_i >$ sent by S_{B2} is routed by GDV to a switch closest to the regional location $H(k_i)$ (labeled by S_{r2} in the figure) inside S_{B2}'s access network. S_{r2} then becomes a global resolver of host i.

To discover the key-value tuple $< k_i, v_i >$ of host i, a switch S_j first sends a lookup message to location $H(k_i)$ in its region. The lookup message arrives at a switch S_u closest to $H(k_i)$, as illustrated in Figure 4.4 (upper left). If S_j and host i were in the same region, S_u would be the regional resolver of i and it would reply to S_j with the key-value tuple of host i. Given that S_j and host i are in different regions, it is very unlikely that S_u happens to be a global resolver of host i (however the probability is nonzero). If S_u cannot find host i's tuple in its local memory, it forwards the lookup message to one of the border switches in its region, S_{B3} in Figure 4.4. Then S_{B3} computes location $H(k_i)$ in the backbone virtual space and includes it with the lookup message, which is routed by GDV to the border switch S_{B2} closest to $H(k_i)$. In the scenario illustrated in Figure 4.4, S_{B2} is not host i's global receiver and it forwards the lookup message to switch S_{r2} closest to the regional location $H(k_i)$, which is the global resolver of host i.

In the above examples, the core and access networks use different virtual spaces but they all use the same hash function H. We note that different hash functions can be used in different networks. It is sufficient that all switches in the same network (access or core) agree on the same hash function, just like they must agree on the same virtual space.

Chapter 5

Performance Evaluation of ROME

5.1 Methodology

The ROME architecture and protocols have been designed with the objectives of *scalability*, *efficiency*, and *reliability*. ROME was evaluated using a packet-level event-driven simulator in which ROME protocols as well as the protocols, GDV, VPoD, and MDT [29, 46] used by ROME are implemented in detail. Every protocol message is routed and processed by switches hop by hop from source to destination. Since our focus is on routing protocol design, queueing delays at switches were not simulated. Packet delays from one switch to another on an Ethernet link are sampled from a uniform distribution in the interval $[50\ \mu s, 150\ \mu s]$ with an average value of 100 μs. This abstraction, aside from speeding up simulation runs, allows performance evaluation and comparison of routing protocols unaffected by congestion issues. The same abstraction was used in the packet-level simulator of SEATTLE [25].

For comparison with ROME, we implemented SEATTLE protocols in detail in our simulator. We conducted extensive simulations to evaluate ROME and SEATTLE in large networks and dynamic networks with reproducible topologies. For the link-state protocol used by SEATTLE, we use OSPF [41] in our simulator. The default OSPF link state broadcast frequency is once every 30 seconds. Therefore, in ROME, each switch runs the MDT maintenance protocol once every 30 seconds.

In ROME, a host's key-value tuple may be published using one location hash or two location hashes. In the case of publishing two location hashes for each tuple, the area of the second hash region is 1/4 of the entire virtual space.

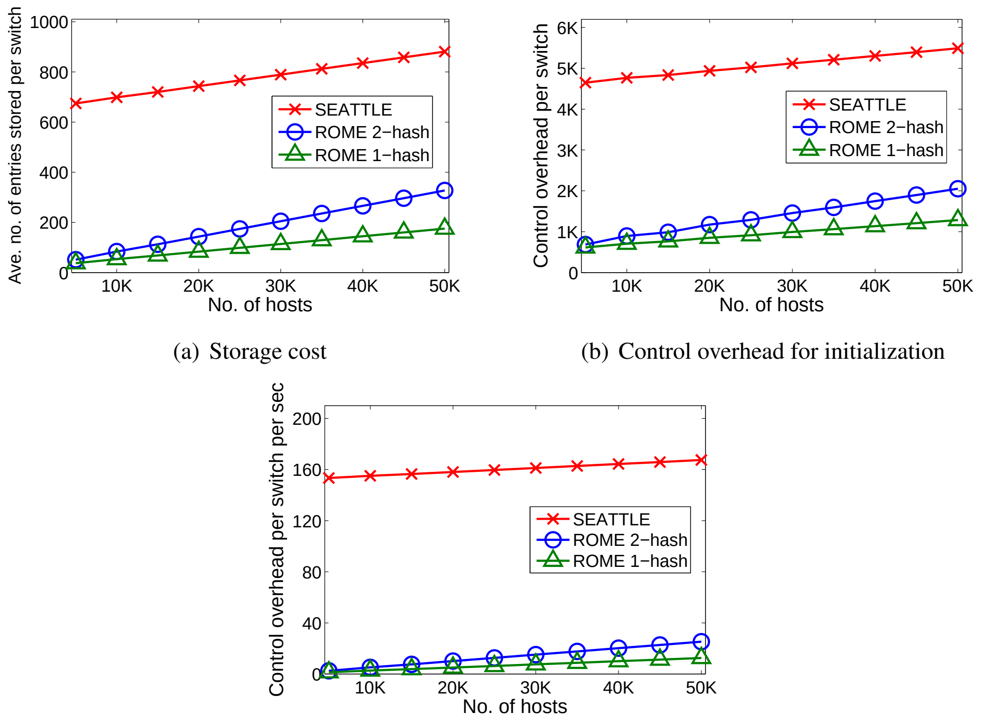
Performance criteria. *Storage cost* is measured by the *average number of entries stored per switch.* These entries include forwarding table entries and host information entries (key-value tuples).

Control overhead is communication cost measured by the *average number of control message transmissions*, for three cases: (i) network initialization, (ii) network in steady state, and (iii) network under churn. Control overhead of ROME for initialization includes those used by switches to determine virtual locations using VPoD, construct a multi-hop DT using MDT protocols, and populate the D^{2}HT with host information for all hosts. Control overhead of SEATTLE for initialization includes those used by switches for link-state broadcast and to populate the one-hop DHT with host information for all hosts. During steady state (also during churn), switches in SEATTLE and ROME use control messages to detect inconsistencies in forwarding tables as well as key-value tuples stored locally and externally. Extra control messages are used to repair inconsistencies in forwarding tables and key-value tuples to recover from churn.

We measure two kinds of *latencies to deliver ROME packets*: (i) latency of the first packet to an unknown host, which includes the latency for host discovery, and (ii) latency of a packet to a discovered host.

To evaluate ROME's (also SEATTLE's) resilience under churn, we show the *routing failure rates* of first packets to unknown hosts and packets to discovered hosts. Successful routing of the first packet to an unknown host requires successful host discovery as well as successful packet delivery by switches from source to destination.

Network topologies used. The first set of experiments used the AS-6461 topology



(a) Storage cost

(b) Control overhead for initialization

(c) Control overhead in steady state

Figure 5.1: Performance comparison by varying the number of hosts

with 654 routers from Rocketfuel data [53] where each router is modeled as a switch. To evaluate the performance of ROME as the number of switches increases, synthetic topologies generated by BRITE with the Waxman model [40] at the router level were used. Every data point plotted in Figures 5.2, 5.3, and 5.5 is the average of 20 runs from different topologies generated by BRITE. *Upper and lower bars in the figure show maximum and minimum values of each data point* (these bars are omitted in Figure 5.3(c) for clarity). Most of the differences between maximum and minimum values in these figures are very small (many not noticeable) with the exception of latency values in Figures 5.3(a) and (b).

5.2 Varying the number of hosts

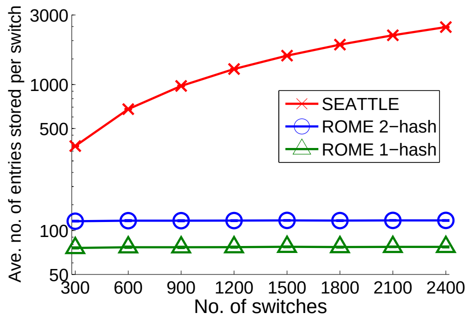
For a network with n switches and m hosts, a conventional Ethernet requires $O(nm)$ storage per switch while SEATTLE requires $O(m)$ storage per switch. We found that ROME also requires $O(m)$ storage per switch with a smaller absolute value than that of SEATTLE. We performed simulation experiments for a fixed topology (AS-6461) with 654 switches. The number of hosts at each switch varies. The total number of hosts of the entire network varies from 5,000 to 50,000. We found that the storage costs of ROME and SEATTLE for forwarding tables are constant, while their storage costs for host information increase linearly as the number of hosts increases. In Figure 5.1(a), the difference between the storage costs of ROME and SEATTLE is the difference in their forwarding table storage costs per switch. The host information storage cost of ROME using two (location) hashes is close to, but not larger than, twice the storage cost of ROME using one hash.

Figures 5.1(b) and 5.1(c) show the control overheads of ROME and SEATTLE, for initialization and in steady state. We found that the control overheads for constructing and updating SEATTLE's one-hop DHT and ROME's D^{2}HT both increase linearly with m and they are about the same. However, the figures show that ROME's overall control overhead is much smaller than that of SEATTLE. This is because ROME's forwarding table construction and maintenance are flooding-free and thus much more efficient.

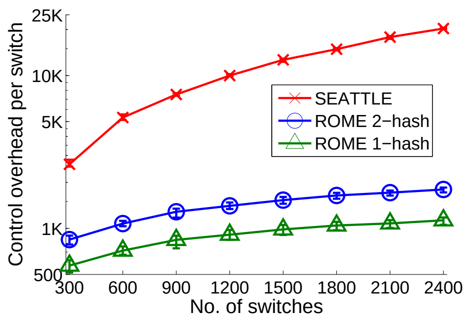
5.3 Varying the number of switches

In this set of experiments the number n of switches increases from 300 to 2,400 while the average number of hosts per switch is fixed at 20. Thus the total number of hosts of the network also increases linearly from 6,000 to 48,000. The results are shown in Figure 5.2. Note that each y-axis is in logarithmic scale.

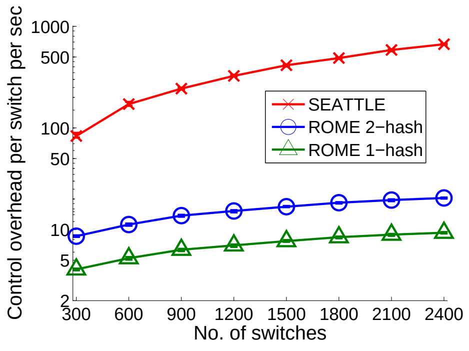
Figure 5.2(a) shows storage cost versus n. Note that while the storage cost of SEATTLE increases with n, ROME's storage cost is almost flat versus n. At $n = 2400$, ROME's



(a) Storage cost



(b) Control overhead for initialization



(c) Control overhead in steady state

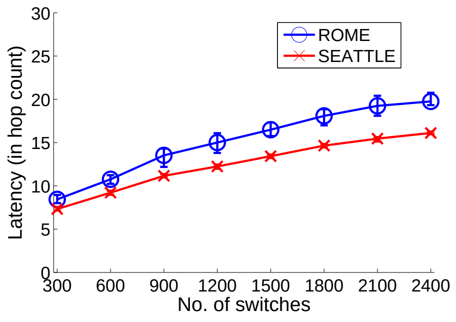
Figure 5.2: Performance comparison by varying the number of switches

storage cost is less than 1/20 of the storage of SEATTLE.

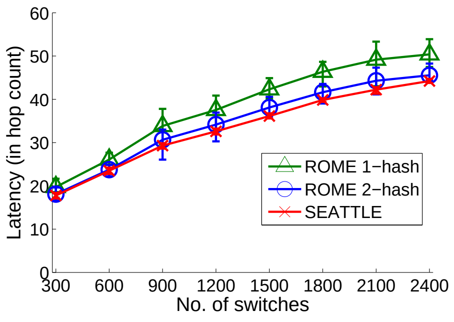
Figures 5.2(b) and (c) show that the control overheads of ROME for initialization and in steady state are both substantially lower than those of SEATTLE. These control overheads of ROME increase slightly with n. This is because the paths from publishers to resolvers in a larger network are longer.

5.4 Routing latencies

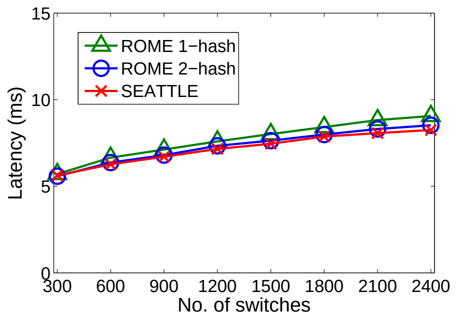
These experiments were performed using the same network topologies (with 20 hosts per switch on average) as in Subsection 5.3. Figure 5.3(a) shows the latency (in average number of hops) of packets to discovered hosts. Note that ROME's latency is not much higher than



(a) Packet to a discovered host



(b) 1st packet to an unknown host



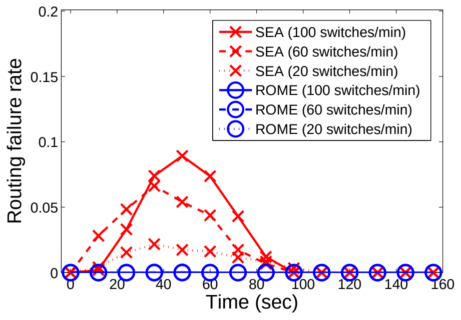
(c) 1st packet to unknown host in hybrid nets

Figure 5.3: Latency vs. number of switches

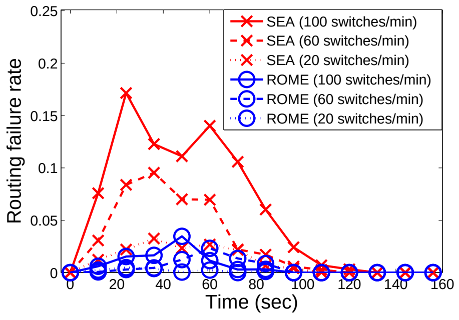
the shortest-path latency of SEATTLE.

Figure 5.3(b) shows the latency of first packets to unknown hosts for SEATTLE and for ROME using one and two hashes. This latency includes the round-trip delay between sender and resolver, and the subsequent latency from sender to destination. By using two hashes instead of one, the latency of ROME improves and becomes very close to the latency of SEATTLE. At $n = 300$, the latency of ROME (2-hash) is actually smaller than the latency of SEATTLE.

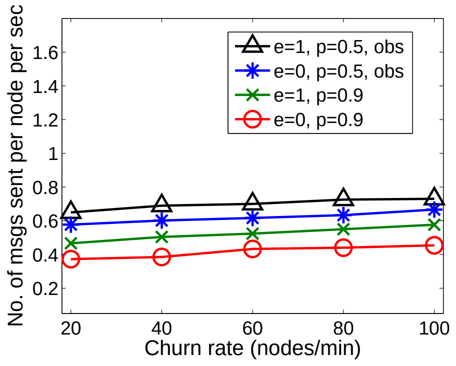
We also performed experiments to evaluate ROME and SEATTLE latencies in hybrid networks, where 20% of the switches are replaced by wireless switches. The packet delay of a wireless hop is sampled uniformly from $[5\ ms, 15\ ms]$ with an average value of $10\ ms$, much higher than $100\ \mu s$ for a wired connection. Figure 5.3(c) shows that SEATTLE



(a) Routing failure rate to a discovered host



(b) Routing failure rate to an unknown host



(c) Control overhead

Figure 5.4: Performance under network dynamics

still has the lowest latency, but the difference between SEATTLE and ROME is negligible.

5.5 Resilience to network dynamics

We performed experiments to evaluate the resilience of ROME using two hashes and SEATTLE under network dynamics for networks with 1,000 switches and 20,000 hosts. Before starting each experiment, consistent forwarding tables and DHTs were first constructed. During the period of 0-60 seconds, new switches joined the network and existing switches failed. The rate at which switches join, equal to the rate at which switches fail, is called the churn rate. Figure 5.4(a) shows the routing failure rates to discovered hosts as a function of time for ROME and SEATTLE. Different curves correspond to churn rates of 20, 60, and 100 switches per minute. At these very high churn rates, the routing failure rate of ROME is close to zero. The routing failure rate of SEATTLE is relatively high but it converged to zero after 100 seconds (40 seconds after churn stopped).

Figure 5.4(b) shows routing failure rates to unknown hosts versus time. Both SEATTLE and ROME experienced many more routing failures which include host discovery failures. The routing failure rate of ROME at the churn rate of 100 switches/minute is still less than that of SEATTLE at the churn rate of 20 switches/minute.

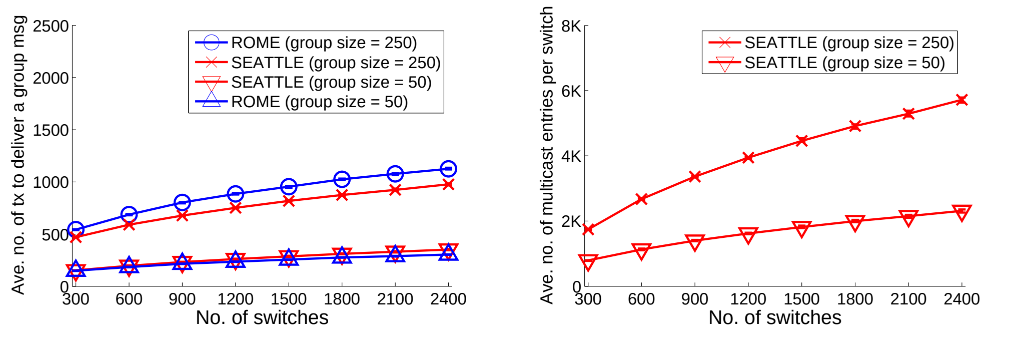
Figure 5.4(c) shows the control overhead (per switch per second) during a period of churn and recovery versus churn rate. The control overhead of SEATTLE is very high due to link-state broadcast. The control overhead of ROME is about two orders of magnitude smaller than that of SEATTLE.

ROME has much smaller routing failure rates and control overhead because each switch (using the MDT maintenance protocol) can find all its neighbors in the multi-hop DT of switches very efficiently without broadcast.

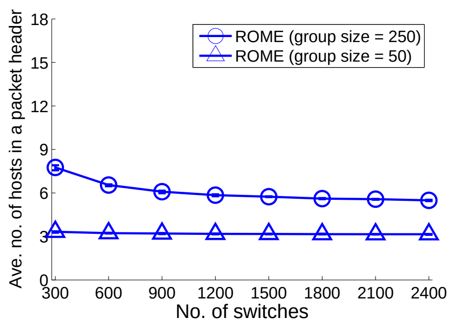
5.6 Performance of multicast

Both SEATTLE and ROME provide multicast support for services like VLAN. SEATTLE uses a multicast tree for each group which requires switches in the tree to store some multicast state. ROME uses the stateless multicast protocol described in Subsection 4.1.3. We performed experiments using the same network topologies (with 20 hosts per switch on average) as in Subsection 5.3. The average multicast group size is 50 or 250 in an experiment. The number of groups is 1/10 of the number of hosts.

Figure 5.5(a) shows the average number of transmissions used to deliver a group message versus the number n of switches. For multicast using a tree, this is equal to the number of links in the tree. SEATTLE used few transmissions than ROME in experiments for average group size 250. ROME used fewer transmissions in experiments for average



(a) Ave. no. of transmissions to deliver a group message (b) Multicast storage cost per switch of SEATTLE

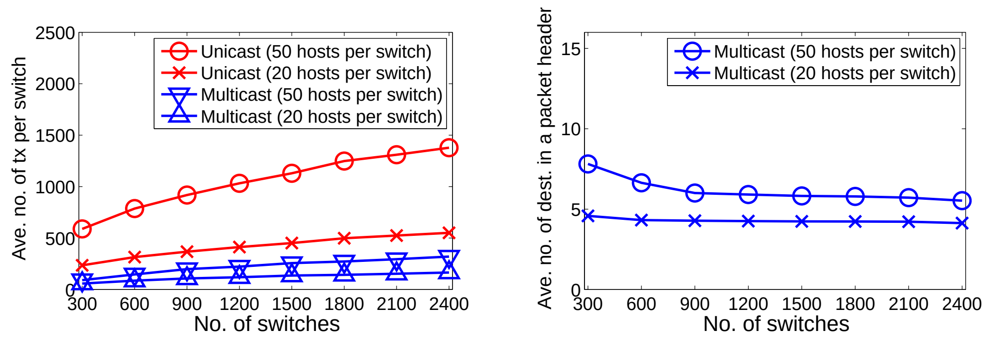


(c) Ave. no. of destinations in the header of a ROME group message

Figure 5.5: Performance of multicast

group size 50.

Figure 5.5(b) shows the amount of multicast state (average number of groups) per switch in SEATTLE versus n. (ROME's multicast is stateless.) Each switch in SEATTLE stores multicast state for a large number of groups, i.e., thousands in these experiments. (Group membership information stored at rendezvous points is not included because it is needed by both ROME and SEATTLE.) On the other hand, ROME requires the packet header of each group message to store a subset of hosts in the group. (SEATTLE does not have this overhead.) Figure 5.5(c) shows the average number of hosts in a ROME packet header. For experiments in which average group size is 50, the number is around 3. For experiments in which average group size is 250, the number is about 6.



(a) Ave. no. of transmissions to publish or update tuples (b) Ave. no. of destinations in a multicast packet header

Figure 5.6: Using multicast to publish or update tuples

We evaluated the performance of using multicast to publish or update key-value tuples. Suppose every switch publishes (or refreshes) its local tuples at the same time. The average number of hosts for each switch is 20 or 50 in an experiment, and the number of hash functions is 1. Figure 5.6(a) shows the average number of transmissions used to publish (or refresh) tuples versus the number n of switches by unicast/multicast. When the average number of hosts per switch is 20, multicast provides about 70% reduction in the number of transmissions used by unicast. When the average number of hosts per switch is 50, multicast provides about 80% reduction in the number of transmissions used by unicast. Figure 5.6(b) shows the average number of destinations in a multicast packet header. The average number is about 5 in the case of 20 hosts per switch and about 6 in the case of 50 hosts per switch.

5.7 Performance of a very large hierarchical network

We use a hierarchical network consisting of 25 access networks of 1000 switches each (generated by BRITE at router level). Two switches in each access network serve as border switches in a backbone network of 50 switches with topology generated by Brite at AS level. Kim et al. [25] discussed ideas for a multi-level one-hop DHT. Based upon the discussion,

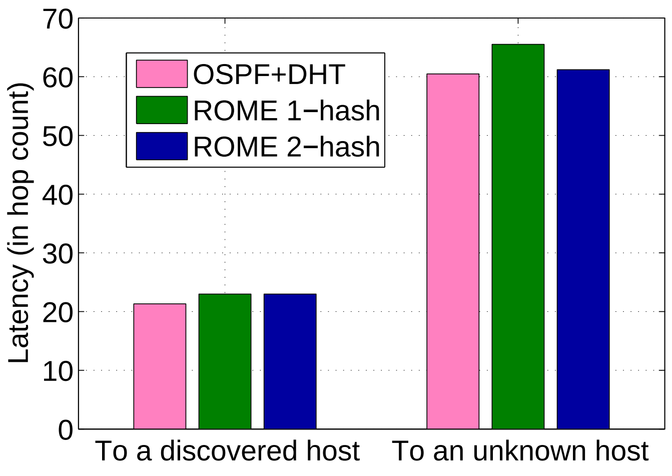
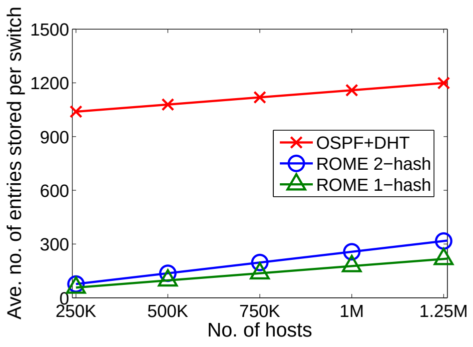


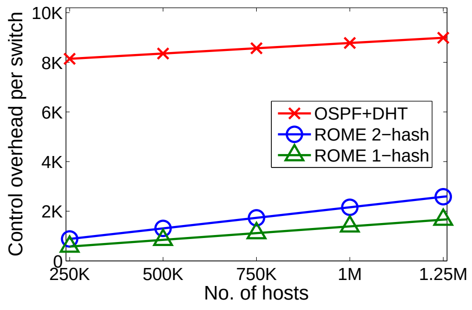
Figure 5.7: Latency comparison for a very large hierarchical network

we implemented in our packet-level event-driven simulator an extension to SEATTLE for routing in a hierarchical network, which we refer to as ”OSPF+DHT”.

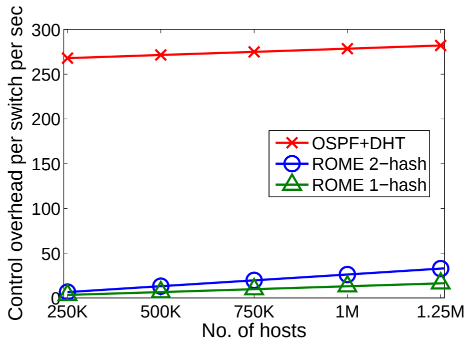
We performed experiments for this network of 25,000 switches for 250K to 1.25 million hosts. Figure 5.7 shows the routing latencies for ROME and OSPF+DHT. ROME's latency to a discovered host is very close to the shortest-path latency of OSPF+DHT, much closer than the latencies in single-region experiments shown in Figure 5.3(a). ROME's latency to an unknown host is also very close to the shortest-path latency of OSPF+DHT. Figure 5.8 shows the storage cost per switch, control overheads for initialization and in steady state. The performance of ROME is about an order of magnitude better than the OSPF+DHT approach.



(a) Storage cost



(b) Control overhead for initialization



(c) Control overhead in steady state

Figure 5.8: Performance comparison for a very large hierarchical network

Chapter 6

Summary and Future Work

6.1 Summary

Large-scale networks such as Metro Ethernet and data center networks, bring new challenges to current network infrastructures. In traditional routing techniques, such as spanning tree or shortest-path routing, each node, such as a switch, router, or wireless sensor, is required to store a large amount of routing state proportional to the network size. However, node memory is hard to scale. For example, high-speed memory for switch forwarding tables is expensive and power-intensive. Furthermore, spanning tree or shortest path routing is not resilient to network dynamics.

In my dissertation research, we focus on designing network infrastructure and protocols for scalable, resilient, and self-managing layer-2 networks. We started by considering replacing shortest-path routing protocols which are expensive in both storage and control overhead, with a more efficient and scalable routing technique. We proposed to use greedy routing that makes routing decision based on node virtual locations. Greedy routing does not always choose the shortest path, but is scalable because the routing state at each node is independent of network size. Moreover, in my protocols, failure recovery at each node under churn is limited to a small subset of all nodes. Prior to my work, greedy routing is generally considered applicable to only wireless networks with several unrealistic assumptions. We discovered how to make greedy routing technique work in any layer-2 network including Ethernet and wireless networks.

We solved a number of core problems in designing this network infrastructure:

1. *Scalable and resilient routing*: We design MDT protocols that provide guaranteed delivery on a multi-hop DT. The guaranteed delivery property is proved for any connected network topology, with node locations specified by any coordinates in a Euclidean space. MDT protocols have fast convergence, strong resilience, and scalability in both storage and control overheads.

2. *Addressing by virtual positioning*: Although MDT provides delivery guarantee for nodes with arbitrary coordinates, greedy routing finds near-optimal paths only when the virtual distance between two nodes reflects the actual routing cost. We designed VPoD, the first network virtual positioning protocol in layer 2. VPoD enables each node to compute a position in a virtual space for itself, such that the Euclidean distance between any pair of nodes in the virtual space is a good estimate of the routing cost between them, for any additive metric. GDV is a greedy routing protocol on VPoD coordinates that achieves both scalability of greedy routing and near-optimal routing path.

3. *Scalable group communication*: We present a stateless multicast protocol for group-wide broadcast or multicast in ROME. A group message is delivered using the locations of its receivers without construction of any multicast tree. Switches do not store any state for delivering group messages. Stateless multicast provides switch memory scalability for group communication in large Ethernet-based networks.

4. *Efficient host discovery*: Traditional name services use either a central name server such as DNS, or flooding such as conventional Ethernet. We propose to use D^{2}HT for host discovery which has two major advantages: (i) uniformly distributing the storage

cost of host information over all network switches, and (ii) enabling information retrieval by unicast rather than flooding.

To demonstrate scalability, we provide simulation performance results for ROME networks with up to 25,000 switches and 1.25 million hosts. Experimental results show that ROME protocols are efficient and scalable. ROME protocols are highly resilient to network dynamics and its switches quickly recover after a period of churn. The routing latency of ROME is only slightly higher than the shortest-path latency.

6.2 Future Work

Building on the current research work of improving the scalability and resilience of large network infrastructures, we plan to expand our research in several directions. We will investigate the requirements of recent and future network applications, and address new research problems that emerge in the corresponding network infrastructures.

Some possible research topics include:

Software-defined networking (SDN): SDN is a new approach to build scalable and easy-to-manage networks to handle big data collection, processing and storage. Using a centralized controller, SDN simplifies the control plane management of a network. However, SDN also brings new challenges, such as single point of failure and extra traffics to/from the controller. Moreover, SDN does not solve the scalability problem of the data plane, especially when switches/routers have to store a big routing table for hosts and VLANs. On the other hand, the ROME architecture is completely distributed and provides scalability in both control plane and data plane, which is desired for very large networks. I will investigate the features of real networks to find optimal designs between fully centralized and fully distributed approaches.

Building efficient and reliable network infrastructures for ecosystems supporting big data analytics: Distributed systems and networks play two important roles in increasing the potential of big data: (1) collecting, transferring, and storing data, and

(2) delivering immediate actions on data insights. Many system and network characteristics significantly affect the performance of big data analytics, such as availability and resiliency, burst handling and queuing, oversubscription, data node bandwidth, and network latency. Each of these characteristics brings along new research issues, and choosing the right trade-off between them is also crucial.

Scalable multicast for large-scale data communication: Multicast is a crucial technique to make efficient use of the available network bandwidth of data center networks, which is used by many applications such as MapReduce and file replication. Due to limited switch memory and the large number of multicast groups, IP multicast that requires every switch to store multicast state is not scalable. Multicast based on the use of Bloom filter has also been proposed, but false positives cause traffic leakage and routing loops [37]. The stateless multicast protocol proposed in ROME [47] is a scalable solution. One potential problem is that the extra storage in packet headers increases the bandwidth overhead. We are designing a multicast protocol that has no extra storage in packet headers, while requiring very few switches to store multicast state. The multicast packet is delivered by recursive unicast at a number of levels. Multicast state is only stored at a few switches called split points (SP), which forward the packet to SPs of the next level. Intermediate switches between SPs are free of multicast state. The main challenge of this design is how to determine the SPs at each level.

Bibliography

[1] Metro ethernet. http://metro-ethernet.org/ .

[2] Understanding VLAN Trunk Protocol (VTP). Cisco Technical Support & Documentation, 2007.

[3] Big Data in the Enterprise: Network Design Considerations. Cisco White Paper, 2011.

[4] S. M. N. Alam and Z. J. Haas. Coverage and Connectivity in Three-Dimensional Networks. In *Proc. of ACM Mobicom*, 2006.

[5] AT&T. Metro ethernet service. http://www.business.att.com/enterprise/Service/network-services/ethernet/metro-gigabit/ .

[6] AT&T. Wide area ethernet. http://www.business.att.com/enterprise/Service/network-services/ethernet/wide-area-vpls/ .

[7] P. Bose and P. Morin. Online routing in triangulations. *SIAM journal on computing*, 33(4):937–951, 2004.

[8] P. Bose, P. Morin, I. Stojmenovic, and J. Urrutia. Routing with Guaranteed Delivery in Ad Hoc Wireless Networks. In *Proc. of the International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications (DIALM)*, 1999.

[9] M. Caesar, M. Castro, E. B. Nightingale, G. O'Shea, and A. Rowstron. Virtual Ring

Routing: Networking Routing Inspired by DHTs. In *Proceedings of ACM Sigcomm*, 2006.

[10] A. Caruso, S. Chessa, S. De, and R. Urpi. GPS Free Coordinate Assignment and Routing in Wireless Sensor Networks. In *Proceedings of IEEE INFOCOM*, pages 150–160, 2005.

[11] J. C. Corbett et al. Spanner: Google's Globally-Distributed Database. In *Proceedings of USENIX OSDI*, 2012.

[12] D. S. J. D. Couto, D. Aguayo, J. Bicket, and R. Morris. A High-Throughput Path Metric for Multi-Hop Wireless Routing. In *Proceedings of ACM MobiCom*, 2003.

[13] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A Decentralized Network Coordinate System. In *Proceedings ACM SIGCOMM*, 2004.

[14] R. Draves, J. Padhye, and B. Zill. Routing in Multi-radio, Multi-hop Wireless Mesh Networks. In *Proceedings of ACM Mobicom*, 2004.

[15] R. Droms. Dynamic Host Configuration Protocol. RFC 2131, 1997.

[16] S. Durocher, D. Kirkpatrick, and L. Narayanan. On Routing with Guaranteed Delivery in Three-Dimensional Ad Hoc Wireless Networks. In *Proceedings of ICDCN*, 2008.

[17] R. Flury and R. Wattenhofer. Randomized 3D Geographic Routing. In *Proceedings of IEEE Infocom*, 2008.

[18] R. Fonseca, S. Ratnasamy, J. Zhao, C. T. Ee, D. Culler, S. Shenker, and I. Stoica. Beacon-Vector Routing: Scalable Point-to-Point Routing in Wireless Sensor Networks. In *Proc. of NSDI*, 2005.

[19] S. Fortune. Voronoi diagrams and Delaunay triangulations. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*. CRC Press, second edition, 2004.

[20] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: a scalable and flexible data center network. In *Proceedings of ACM SIGCOMM*, 2009.

[21] S. Halabi. *Metro Ethernet.* Cisco Press, 2003.

[22] M. Huynh and P. Mohapatra. Metropolitan Ethernet Network: A Move from LAN to MAN. *Computer Networks*, 51, 2007.

[23] S. Jain, Y. Chen, S. Jain, and Z.-L. Zhang. VIRO: A Scalable, Robust and Name-space Independent Virtual Id ROuting for Future Networks. In *Proc. of IEEE INFOCOM*, 2011.

[24] B. Karp and H. Kung. Greedy Perimeter Stateless Routing for Wireless Networks. In *Proceedings of ACM Mobicom*, 2000.

[25] C. Kim, M. Caesar, and J. Rexford. Floodless in SEATTLE: A Scalable Ethernet Architecture for Large Enterprises. In *Proc. of Sigcomm*, 2008.

[26] C. Kim and J. Rexford. Revisiting Ethernet: Plug-and-play made scalable and efficient. In *Proceedings of IEEE LAN/MAN Workshop*, May 2007.

[27] Y.-J. Kim, R. Govindan, B. Karp, and S. Shenker. Geographic Routing Made Practical. In *Proceedings of USENIX NSDI*, 2005.

[28] S. S. Lam and C. Qian. Geographic Routing in d-dimensional Spaces with Guaranteed Delivery and Low Stretch. Technical Report TR-10-03, The Univ. of Texas at Austin, Dept. of Computer Science, 2010.

[29] S. S. Lam and C. Qian. Geographic Routing in d-dimensional Spaces with Guaranteed Delivery and Low Stretch. In *Proceedings of ACM SIGMETRICS*, June 2011, accepted for publication in *IEEE/ACM Transactions on Networking*.

[30] D.-Y. Lee and S. S. Lam. Protocol design for dynamic Delaunay triangulation. Technical Report TR-06-48, The Univ. of Texas at Austin, Dept. of Computer Science, December 2006.

[31] D.-Y. Lee and S. S. Lam. Protocol design for dynamic Delaunay triangulation. Technical Report TR-06-48, The Univ. of Texas at Austin, Dept. of Computer Science, December 2006; an abbreviated version in *Proceedings IEEE ICDCS*, June 2007.

[32] D.-Y. Lee and S. S. Lam. Efficient and Accurate Protocols for Distributed Delaunay Triangulation under Churn. In *Proceedings of IEEE ICNP*, November 2008.

[33] E. A. Lee. Cyber physical systems: Design challenges. Technical Report UCB/EECS-2008-8, EECS Department, University of California, Berkeley, Jan 2008.

[34] S. Lee, B. Bhattacharjee, and S. Banerjee. Efficient Geographic Routing in Multihop Wireless Networks. In *Proceedings of ACM MobiHoc*, 2005.

[35] B. Leong, B. Liskov, and R. Morris. Geographic Routing without Planarization. In *Proceedings of USENIX NSDI*, 2006.

[36] B. Leong, B. Liskov, and R. Morris. Greedy Virtual Coordinates for Geographic Routing. In *Proceedings of IEEE ICNP*, 2007.

[37] D. Li et al. Scalable Data Center Multicast using Multi-class Bloom Filter. In *Proceedings of IEEE ICNP*, 2011.

[38] M. Li and Y. Liu. Rendered Path: Range-free Localization in Anisotropic Sensor Networks with Holes. In *Proceedings of ACM Mobicom*, 2007.

[39] Y. Liu, L. M. Ni, and M. Li. A Geography-free Routing Protocol for Wireless Sensor Networks. In *Proceedings of HPSR*, 2005.

[40] A. Medina, A. Lakhina, I. Matta, , and J. Byers. BRITE: An Approach to Universal Topology Generation. In *Int. Workshop on Modeling, Analysis and Simulation of Computer and Telecom. Systems*, 2001.

[41] J. Moy. OSPF Version 2. RFC 2328, 1998.

[42] A. Myers, T. E. Ng, and H. Zhang. Rethinking the service model: Scaling ethernet to a million nodes. In *Proceedings of HotNets*, 2004.

[43] T. S. E. Ng and H. Zhang. Predicting Internet network distance with coordinates-based approaches. In *Proceedings of INFOCOM*, 2002.

[44] D. Plummer. An Ethernet Address Resolution Protocol. RFC 826, 1982.

[45] D. Pompili, T. Melodia, and I. Akyildiz. Routing algorithms for delay-insensitive and delay-sensitive applications in underwater sensor networks. In *Proc. 12th Int. Conf. on Mobile Computing and Networking*, 2006.

[46] C. Qian and S. S. Lam. Greedy Distance Vector Routing. In *Proceedings of IEEE ICDCS*, June 2011.

[47] C. Qian and S. S. Lam. ROME: Routing On Metropolitan-scale Ethernet. In *Proceedings of IEEE ICNP*, 2012.

[48] A. Rao, S. Ratnasamy, C. Papadimitriou, S. Shenker, and I. Stoica. Geographic Routing without Location Information. In *Proceedings of ACM Mobicom*, 2003.

[49] S. Ray, R. Guerin, and R. Sofia. A distributed hash table based address resolution scheme for large-scale Ethernet networks. In *Proceedings of Int. Conf. on Communications*, June 2007.

[50] R. Rivest. The MD5 Message-Digest Algorithm. RFC 1321, 1992.

[51] D. Sampath, S. Agarwal, and J. Garcia-Luna-Aceves. Ethernet on AIR: Scalable Routing in Very Large Ethernet-based Networks. In *Proceedings of IEEE ICDCS*, 2010.

[52] K. Seada, M. Zuniga, A. Helmy, and B. Krishnamachari. Energy-efficient forwarding strategies for geographic routing in lossy wireless sensor networks. In *Proceedings of ACM SenSys*, 2004.

[53] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP Topologies with Rocketfuel. In *Proceedings of ACM SIGCOMM*, 2002.

[54] M.-J. Tsai, H.-Y. Yang, B.-H. Liu, and W.-Q. Huang. Virtual-Coordinate-Based Delivery-Guaranteed Routing Protocol in Wireless Sensor Networks. *IEEE/ACM TRANSACTIONS ON NETWORKING*, 17, 2009.

[55] Y. Zhao, Y. Chen, B. Li, and Q. Zhang. Hop ID: A Virtual Coordinate based Routing for Sparse Mobile Ad Hoc Networks. *IEEE Transaction on Mobile Computing*, 2007.

[56] J. Zhou, Y. Chen, B. Leong, and P. Sundaramoorthy. Practical 3D Geographic Routing for Wireless Sensor Networks. In *Proceedings of Sensys*, November 2010.