SmartBridge: A Scalable Bridge Architecture

Thomas L. Rodeheffer
Systems Research Center, Compaq Computer Corporation
130 Lytton Ave, Palo Alto, CA 94301
tom.rodeheffer@compaq.com

Chandramohan A. Thekkath
Systems Research Center, Compaq Computer Corporation
130 Lytton Ave, Palo Alto, CA 94301
chandu.thekkath@compaq.com

Darrell C. Anderson
Computer Science Department, Duke University
Levine Science Research Center, Durham, NC 27708
anderson@cs.duke.edu

ABSTRACT

As the number of hosts attached to a network increases beyond what can be connected by a single local area network (LAN), forwarding packets between hosts on different LANs becomes an issue. Two common solutions to the forwarding problem are *IP routing* and *spanning tree bridging*. IP routing scales well, but imposes the administrative burden of managing subnets and assigning addresses. Spanning tree bridging, in contrast, requires no administration, but often does not perform well in a large network, because too much traffic must detour toward the root of the spanning tree, wasting link bandwidth.

This paper introduces a new architecture, called *SmartBridge*, that combines the good features of IP routing and spanning tree bridging. We have implemented the SmartBridge design for 10 Mb/s and 100 Mb/s Ethernet LANs, using standard PC hardware with off-the-shelf network interface cards and running our algorithms in software. Our 100 Mb/s system runs at full link bandwidth.

1. INTRODUCTION

A single local area network (LAN) can connect a limited number of hosts over a limited geographical extent. Interconnecting a large collection of hosts that are not in close proximity usually requires the use of multiple LANs and some arrangement to forward packets between hosts on different LANs.

Two common arrangements in modern networks are *IP routing* and *spanning tree bridging*. IP routing works at the network level between hosts that use the IP protocol. Spanning tree bridging works at the link level and is independent of the network protocol (IP or otherwise) used by the hosts.

IP routing scales well and is a good solution when the network installation is managed by a competent authority who can administer the required subnet and host address assignments. If, however, authorities are vague or overlapping, such as, for example, often happens between various departments of a large organization, then an arrangement that does not require much administration may be more suitable.

Spanning tree bridges, such as those that follow the IEEE 802.1 standard [9], have several good properties.

1. Spanning tree bridges have algorithms built into them that make them truly self-configuring and transparent to the hosts. Spanning tree bridges require no manual configuration nor do they require any configuration information in the hosts that are connected to them.
2. Spanning tree bridges are independent of the network protocol and can therefore accommodate a variety of hosts running different network protocols. In particular, spanning tree bridges can forward packets containing protocols that are unroutable at the network level, for example, MOP [6], LAT, and NETBEUI [8].
3. Spanning tree bridges are simple devices; they are therefore inexpensive to build and very robust.

Unfortunately, spanning tree bridges have some disadvantages that preclude them from being more widely deployed. Typically, total inter-LAN traffic increases as the size of the network grows and eventually the bandwidth through some bridge or LAN becomes a bottleneck. Adding more bridges or LANs does not solve the problem, because even if there are redundant connections in the network, only connections that are part of the spanning tree are allowed to carry traffic. Furthermore, routes along the spanning tree tend to be longer—sometimes considerably longer—than shortest paths, and thus packet latency and aggregate utilization is higher than in a system in which all routes are shortest paths. Packet latency remains a problem even in a highly underutilized network. A careful choice of the spanning tree may ameliorate some of these problems in some cases, but in general these problems are unavoidable in a spanning tree bridge system as the size of the network grows.

In this paper, we describe a new bridge design, called *SmartBridge*. Our design takes a systems approach to solving the inter-LAN forwarding problem. Our goal is to retain the good properties of spanning tree bridges while arranging to forward packets along shortest paths.

It is relatively easy to achieve shortest path routing between LANs. Determining which LAN a host is connected to is a harder problem,

and this problem is yet more difficult because a host may occasionally move from one LAN to another.

Our solution is based on selecting a set of routes that satisfies the following properties: (1) each route is a shortest path, (2) the union of all routes starting at any given LAN forms a *source tree*, which enables the detection of hosts that have moved, and (3) the union of all routes ending at any given LAN forms a *destination tree*, which enables an efficient implementation of forwarding. Although constructing such a set of routes is the key insight into our design, many other problems must be overcome to provide a complete bridging solution, as will be discussed later.

We have implemented the SmartBridge design for 10 Mb/s and 100 Mb/s Ethernet LANs. Using standard PC hardware with off-the-shelf network interface cards and running our SmartBridge algorithms in software, we are able to forward packets at link speeds.

It is important to consider the relevance of our work in the context of current trends in LANs. In modern switched-Ethernet networks, point-to-point links connect hosts to switches and switches to other switches. Switches run the 802.1 spanning tree algorithm to route packets between them. Often, because of the perceived deficiencies of spanning trees, these switches support other routing schemes, such as IP routing or virtual LANs, thus trading off ease of administration to get claimed better performance. We therefore believe that current Ethernet switches used in local area networks can benefit by running the SmartBridge algorithms. In addition, because of convenience or economic needs, a single Ethernet switch port is often connected to multiple hosts by using a hub, which acts just like a multi-drop Ethernet segment. If it is desirable to allow connecting more than one switch to the same hub, for example to obtain higher reliability or relax administrative oversight, then our solution is relevant. Finally, by far the majority of currently deployed LANs today are still 10 Mb/s non-switched Ethernets.

2. BACKGROUND

This section first describes some of the more important requirements of bridging schemes in an extended LAN. We also describe, at a high level, the working of spanning tree bridges to motivate our SmartBridge design.

In the rest of the paper, we use the term *extended LAN* to denote a network composed of a collection of LANs transparently interconnected by bridges. Each of the individual LANs is called a *segment*. A device connected to a LAN is called a *station*. We assume that each station has a globally unique identifier, such as a 48-bit Ethernet UID. We use the term *port* to refer to the part of a station specific to a particular connection between that station and a segment. A *bridge* is a station that forwards packets between segments so as to make the interconnections in the extended LAN transparent to hosts. Bridges will typically have multiple ports, more than one of which may connect to the same segment. We use the term *host* to denote any station that is not a bridge. A host sends and receives packets to and from other hosts. A host is allowed only one port. A non-bridge machine with multiple network interfaces must be considered as a collection of hosts each with its own UID.

2.1 Bridge Requirements

There are three main requirements of a bridging scheme:

1. Bridges must function without any cooperation from the hosts. In a bridging scheme, hosts exchange packets with each other as if they were connected to the same segment. The bridges must discover everything they know about hosts by listening to the traffic between hosts.

2. Bridges must forward exact duplicates of the original packets. There is no provision for a bridge to add or modify headers to host packets, although a bridge can exchange packets of its own with other bridges. This requirement creates difficulties in avoiding forwarding loops and in learning host locations.

3. Bridges must be self-configuring and self-stabilizing [2]. Bridges must discover compatible bridges and agree on the topology of the network without external intervention. Further, bridges must determine stable, loop-free routes between hosts on different segments. Loop-free routes are important because unlike other forms of forwarding (e.g., IP routing) that use hop counts to discard circulating packets, bridges cannot add or modify headers. Bridging must work correctly in the presence of communication failures as well as misconfigurations and operator errors. Agreeing on a stable state of the network in the presence of such failures is a difficult problem.

A fundamental issue with any bridging scheme is that it must be able to learn where a host is located and be able to detect when the host moves. Learning a host location is difficult because typical LANs are multi-drop fabrics, unlike point-to-point networks where hosts plug directly into switches. In a point-to-point network, such as ATM [12] or Autonet [18], a switch can vouch for the presence of a host because it can distinguish a switch-to-switch link from a switch-to-host link. In multi-drop LANs like Ethernet, a bridge cannot determine that a host is attached to a particular segment merely because it hears packets from the host on the segment. Such a packet could be a perfect duplicate forwarded onto that segment by another bridge on behalf of the original host. Notice that if the Ethernet packet format had an additional field, say a simple hop count field, the problems of circulating packets and host location could be trivially solved.

2.2 Spanning Tree Bridges

Spanning tree bridges meet the requirements described above by executing the IEEE 802.1 spanning tree algorithm [15]. This distributed algorithm can be viewed as constructing a spanning tree over a graph of the network topology. The vertices of this graph represent segments and bridges and the edges represent bridge-to-segment connections. It may be noted that the graph is connected and bipartite. The goal of the algorithm is to select a subset of the connections to form a spanning tree.

Figure 1 depicts an extended LAN containing five segments and three bridges. Figure 2 shows the bipartite topology graph for this network. A spanning tree has been chosen with B1 as the root bridge and the connections S3–B3 and B3–S5 selected as standby. Note that host packets from segment S3 to segment S4 must travel along the spanning tree path (S3, B2, S2, B1, S4) rather than the shortest possible path (S3, B3, S4).

Basically, the spanning tree algorithm works as follows. First, the bridges in the network elect one of their members, called the *root bridge*. Then each bridge other than the root bridge determines its distance (minimum path length in the graph) from the root bridge

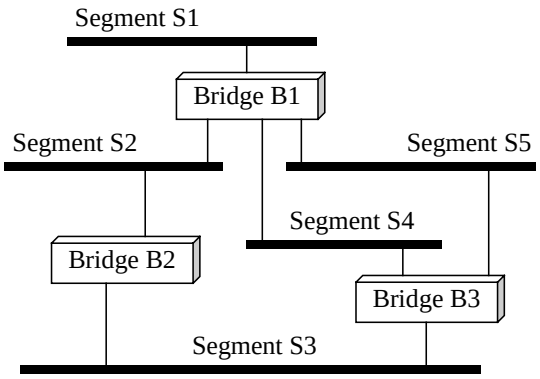

Figure 1: An Extended LAN. *This example network contains five segments interconnected by three bridges.*

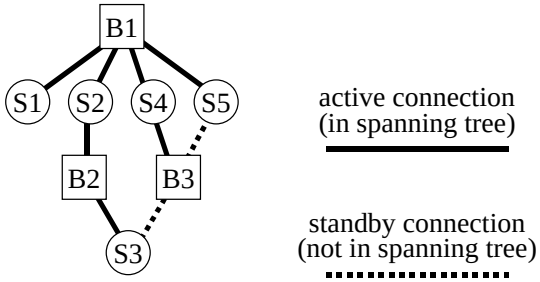

Figure 2: A Network Topology Graph. *The topology graph for the network of Figure 1. Active connections have been selected to form a spanning tree; the remaining connections are standby.*

and selects one of its ports, called the *root port*, that is closest to the root bridge. Then the bridges elect one port on each segment, called the *designated port*, that is closest to the root bridge. The elections and distance determinations are performed by broadcasting packets containing a nomination and applying deterministic tie-breaker rules.

The root ports and designated ports are called *active ports* and the remaining ports in the graph are called *standby ports*. A bridge-to-segment connection is called *active* if its bridge port is active, and likewise for *standby*. The spanning tree consists of all active connections. Since the graph is bipartite, each bridge other than the root has a segment for its parent (connected via its root port) and each segment has a bridge for its parent (connected via its designated port).

Host packets are forwarded only along active connections; this is enforced by dropping host packets that are received via standby ports. Since the set of active connections forms a spanning tree, each pair of segments is connected by a unique loop-free path.

When a bridge receives a multicast packet via an active port, it forwards it via all of its other active ports. This process is called *flooding* and it guarantees that the packet will appear once on each segment in the network and thus is guaranteed to reach its intended destinations.

Flooding is clearly wasteful of network resources for unicast packets, so spanning tree bridges have a provision for learning host locations. When a bridge receives a packet via an active port P, it infers that the packet's source host H can be reached by sending via port P. The bridge records this fact so that it will forward subsequent packets destined to H only via port P. Over time, each bridge learns about the location of each host, to the extent that it knows which active port is to be used to reach each previously seen source host. Since almost all LAN protocols generate at least occasional replies from a destination host, there is a good chance that a packet's destination location will be known. A packet whose destination location is unknown is treated as a multicast and flooded.

If a host moves, its new location will be learned when it sends a packet. In any event, learned locations expire after a while unless refreshed. Thus, spanning tree bridges cope with changes in host location.

3. OUR SOLUTION

The most important difference between SmartBridges and spanning tree bridges is that SmartBridges forward packets between hosts of known location along a shortest possible path in the network. Therefore SmartBridges avoid the congestion and latency problem that results from forcing traffic to detour toward the root of a spanning tree.

SmartBridges operate based on knowing a complete description of the interconnection of bridges and segments in the network. This description is called the network topology graph, G. The graph G is kept current by the processes of *inventory construction* and *topology acquisition* described below. Given G, it is a simple matter for each SmartBridge to compute a shortest path between each pair of segments. We select a special set of shortest paths, called the *best paths set*, that have certain properties described below.

In order to forward packets along shortest paths between hosts, SmartBridges learn the exact segment to which a host connects. A host's location is discovered by a neighboring SmartBridge when the host first transmits a packet. An update protocol informs all SmartBridges of the location of the host.

3.1 The Distributed Systems Problem

Since the SmartBridges form a distributed system, some technique must be used to ensure that the SmartBridges' knowledge of the network topology and host locations converges to a consistent state in the absence of further changes.

One approach to ensure convergence would be to use a link-state protocol [13], in which each bridge regularly emits a packet describing its connections and these packets are flooded to all segments. In the absence of further changes in the network topology, eventually all bridges' knowledge of the network topology will converge. A similar method could be used to ensure convergence in host locations, although a careful design is essential, since detecting a moved host depends on an assumption of how other bridges are forwarding the host's packets.

The main problem with a link-state protocol solution is that the system must tolerate intervals of inconsistency while the protocol is converging. Inconsistency typically produces forwarding loops, which are particularly bad in the case of bridging, since packets have no time-to-live indicator and would consequently circulate until the loop was fixed or some overload detector went off. A link-state protocol also provides no indication of termination, so there is no indication of when the danger of inconsistency might be past.

In contrast, our approach in SmartBridge uses the techniques of diffusing computations [7] and effective global consistency. In a diffusing computation, an initiator sends out requests to its neighbors, who then send requests to their neighbors, and so on, with a reply being returned for each request; if each reply is sent only after the completion of its request, then the initiator will know when the entire computation is complete. Effective global consistency allows different processes to act as if knowledge were consistent everywhere, because a barrier is established that prevents old and new knowledge from coming into contact.

Since we use effective global consistency, forwarding loops cannot happen, although packets may be dropped during changes. Since we use diffusing computations, convergence of the network topology is driven forward efficiently by the protocol and we know when it is complete.

3.2 Inventory Construction

At the very lowest level, the SmartBridge ports connected to a given segment elect (and regularly re-confirm) a designated port. The designated port assigns a globally unique identifier (UID) to the segment and maintains an inventory of the ports connected to the segment. This is the same idea as the LAN Designated Intermediate System in IS-IS [14]. We call this process *inventory construction*.

The inventory construction process is similar to a global membership service [3] such as used in ISIS [4]. However, the goals are different. A global membership service informs each surviving member of the same sequence of membership changes, so that a consistent view of the evolution of the system can be maintained at each stage. In contrast, the inventory construction process does not need to produce consistent intermediate states: its goal is to converge quickly to a state in which each member agrees on the current membership. This is achieved by electing a designated port to announce the current membership.

Although multiple ports on a SmartBridge may be connected to the same segment, this redundancy is not useful for the higher-level functions of topology acquisition and packet forwarding and in fact would confuse host location learning (as explained in §4.3.2). Redundant ports are filtered out of the inventory so that each bridge is represented at most once.

3.3 Topology Acquisition

Any change (addition or removal of bridges) in the filtered inventory initiates an instance of *topology acquisition*, which is a diffusing computation that propagates to all bridges, collects a list of all bridge-to-segment connections, and then distributes this list (which is a representation of the new network topology graph, G) to all bridges. Each instance of topology acquisition is identified by the bridge UID of its initiator and an epoch number. Multiple instances active at the same time compete with each other bridge-by-bridge for control of the network, with the result that the last instance that runs to completion contains the current connections of each bridge.

As topology acquisition propagates to all bridges, it establishes a barrier between any old network topology and the new network topology graph. In effect, SmartBridges react to a change in network topology by rapidly performing a global reboot.

This topology acquisition process was originally developed for Autonet [18], a point-to-point switch-based network. The original process is described in detail in a previous paper [17]. The principal change required for SmartBridge was a modification of the data structures to account the multi-drop nature of segments.

During topology acquisition, we suspend the normal forwarding of packets between hosts. Thus, it is important to complete the topology acquisition process in an acceptably short period of time. With our current implementation, we keep this time on the order of tens of milliseconds, which is short enough that open TCP connections between hosts will resume their normal function after the topology acquisition is complete.

After the network topology graph has been distributed, SmartBridges can learn the location of hosts and forward packets between them using shortest path routing. The following sections describe the learning, forwarding, and routing aspects of our design in more detail.

4. LEARNING AND FORWARDING

SmartBridges forward packets between *hosts* based on shortest paths calculated between *segments*. For this scheme to work correctly, SmartBridges have to know the segments on which hosts are located and the shortest paths between segments. This section describes the process by which SmartBridges learn the location of hosts and how they determine the shortest paths.

At a high level, SmartBridge forwarding is quite straightforward. When a SmartBridge receives a packet, it consults its internal data structures. Depending on the source and destination of the packet, these data structures determine how to forward the packet.

To be concrete, suppose a SmartBridge B receives via one of its connected segments T a packet with source host SH and destination host DH. Figure 3 shows a flowchart of the SmartBridge's actions. Assuming the bridge is not involved in a topology acquisition (§3.3) and is not on a relevant wavefront (§4.2.1), it consults its host location table to determine the segments S and D on which the respective hosts SH and DH are located.

If S is unknown (indicated by $\perp$ in the flowchart), B must drop the packet. In this case, B may trigger a *location revision request* for host SH on segment T, which eventually causes all SmartBridges to learn that host SH is located on segment T, as described below in §4.2.

If S is known, but D is unknown or DH is a multicast address, B floods the packet as described below in §4.3. Such a packet is called a *network flood packet*.

Otherwise, the host location table knows the segments S and D on which the respective hosts SH and DH are located. In this case, B forwards the packet along a shortest path as described below in §4.4. Such a packet is called a *shortest path packet*.

4.1 Consistent Information Guarantee

An important property guaranteed by the SmartBridge design is that every SmartBridge that handles a given host packet applies the same network topology and host location information to the decision of how to handle the packet. Hence, each SmartBridge can handle a packet based on knowing how all other SmartBridges did, or will, handle the packet.

For example, no SmartBridge is allowed to forward any packet whose source segment location it does not know. So if a Smart-

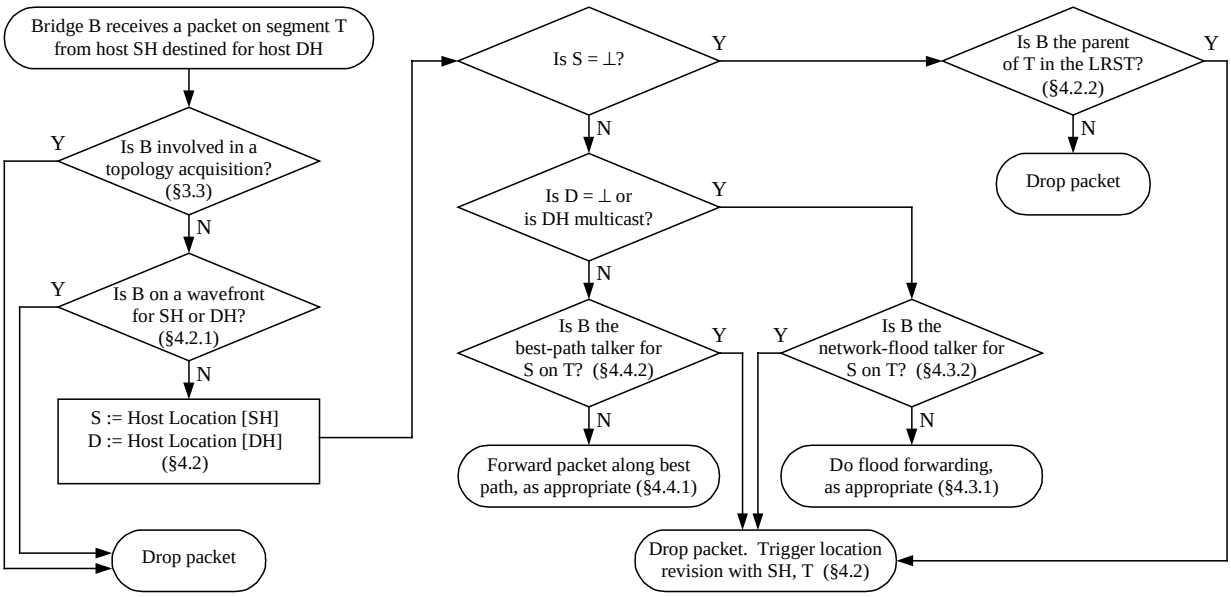

Figure 3: SmartBridge Flowchart for Handling a Host Packet.

Bridge B receives on segment T a packet from source host SH whose segment location is unknown to B, then B knows that every SmartBridge that ever handled this packet also did not know the location of SH, and therefore the packet could not possibly have been forwarded, and therefore the source host SH must be located on segment T.

As another example, when a SmartBridge receives a packet that has a known source location but unknown destination location, it floods this packet on a spanning tree. Since every SmartBridge uses the same host location information and spanning tree when handling the packet, the packet will correctly flood over all segments in the network.

When the consistent information guarantee would otherwise be impossible to maintain, such as across a topology acquisition or a host location revision, SmartBridges drop host packets. During topology acquisition, which is expected to be fairly infrequent, SmartBridges drop all host packets. Host locations are revised using a wavefront protocol that provides an impenetrable barrier between the old host location information and the new. Our wavefront protocol is described in the following section.

4.2 Host Location Revision

Each SmartBridge maintains a host location table that tells which segment a particular host is located on. As hosts are added or moved in the extended LAN, the table can get out of date and will need to be updated. The contents of the host location table on each SmartBridge is managed by the location revision mechanism described below.

To manage host location revision, the SmartBridge with the largest UID in the network is selected as the *location revision root* (LRR). Starting from the LRR, a minimum-depth spanning tree is constructed by performing a deterministic breadth-first traversal of the network topology graph. This tree is called the *location revision spanning tree* ($LRST$). Each SmartBridge performs this computation separately each time a new network topology graph is distributed, but since the computation is deterministic and the distributed graphs are identical, each SmartBridge arrives at the same result.

A SmartBridge triggers a host location revision when it concludes that the location of a host needs to be updated. This can happen in several ways, which will be described shortly. A location revision request consists of a segment identifier and a host UID. Location revision requests are forwarded bridge-by-bridge up the $LRST$ to the LRR, whereupon the LRR initiates a *location revision wavefront* that spreads over the network.

4.2.1 Location Revision Wavefront

A location revision wavefront spreads over the network by SmartBridges exchanging packets with their neighbors according to a wavefront protocol. A wavefront is an instance of a diffusing computation [7] oriented so that it flows over the entire network like a global reset protocol [1]. Except for the fact that it starts at the LRR, the wavefront has nothing to do with the $LRST$.

At any given time, a SmartBridge can be "ahead", "on", or "behind" a particular wavefront. A bridge is ahead as long as it has not received any packets for the wavefront. A SmartBridge is on the wavefront when it receives the first such packet. It remains on the wavefront and sends packets to its neighboring SmartBridges and awaits their acknowledgment. When all neighbors have acknowledged, the SmartBridge is behind the wavefront.

When a location revision wavefront is in progress, we must ensure that packet handling decisions made by SmartBridges on different sides of the wavefront do not violate the consistent information guarantee. Because of the way the wavefront is propagated, any path from a SmartBridge that is ahead of the wavefront to one that is behind the wavefront must pass through a SmartBridge that is on the wavefront. So we require that a SmartBridge on a wavefront regarding host H must drop any packet with source H or destination

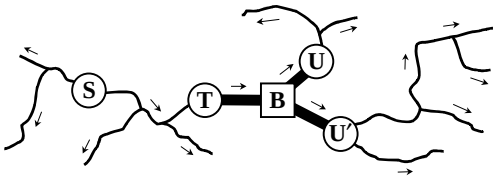

Figure 4: Network Flood Source Tree for S (NFS_S). *Arrows indicate direction away from S. Network flood packets whose source host is known to be located on S are forwarded by B from segment T to segments U and U'. Bridge B is the network flood talker for S on U and U'.*

H and also must drop any location revision request about H. Successive wavefronts are kept in order by using sequence numbers.

It is important to ensure that location revision works correctly in the presence of topology changes. We achieve this by tagging the wavefront packets with a topology acquisition instance identifier (§3.3). A SmartBridge ignores wavefronts that do not belong to its most recent topology acquisition instance.

4.2.2 Detecting Source Host Location

As mentioned above, a SmartBridge triggers a host location revision when it concludes that the location of a host needs to be updated. One situation in which this happens is as follows. Suppose a SmartBridge receives on segment T a packet from a source host SH whose location is unknown. Since no SmartBridge will forward such a packet, it must be the case that host SH is located on T. Although all SmartBridges on T receive the packet and realize that SH must be on T, to avoid redundant work, only the SmartBridge that is the parent of T in the $LRST$ is allowed to trigger a location revision request in this case. The situations when the known location of the source host is in error are described below in §4.3.2 for network flood packets and §4.4.2 for shortest path packets.

4.3 Network Flood Forwarding

A packet with a known source location and a destination that is either a multicast or a host of unknown location is called a *network flood packet*. SmartBridges flood network flood packets along an unrooted spanning tree, called the *network flood spanning tree* ($NFST$).[1] This scheme is quite similar to the IEEE 802.1 bridge standard, but we add a key refinement that enables us to detect and correct errors in the known locations of hosts. We write NFK_S to refer to an arbitrary network flood packet whose source host is known to be on segment S.

For any segment S, the union of all routes along the $NFST$ that proceed away from S forms a directed tree whose source is S. We call this tree NFS_S, the *network flood source tree* for S. Figure 4 provides an illustration.

4.3.1 Forwarding Rule for NFK_S

The *network flood forwarding rule* can be stated as follows: a SmartBridge must forward NFK_S along directed connections that appear in NFS_S. Forwarding must occur along each connection in the proper direction. Supposing SmartBridge B receives NFK_S via segment T, it forwards it onto all segments U such that (T, B, U) appears in NFS_S.

If NFK_S actually starts at segment S, the SmartBridges will flood NFK_S over the $NFST$, since every connection in NFS_S is directed away from S.

4.3.2 Talker Detects a Moved Source Host

Note that any segment $U \neq S$ has a unique parent in NFS_S. We call this parent the *network flood talker* for S on U.

From the forwarding rule, the talker is the only bridge that could possibly forward NFK_S onto U. So if NFK_S appears on U without being forwarded there by the talker, it must be the case that the source host of NFK_S is actually located on segment U, and not on segment S as was believed. This is the *network flood talker check*. Note that the talker and only the talker can evaluate this check for NFK_S on U. The talker triggers a host location revision accordingly.

This scheme requires that a SmartBridge be able to distinguish the packets it transmits from other packets that appear on the segment. No network interface we are aware of receives its own transmissions, so the requirement is satisfied for a single port. However, a SmartBridge can have multiple ports connected to the same segment. We ensure that the requirement is satisfied even in this case by detecting and removing redundant ports during the inventory construction phase, as described in §3.2. One of these redundant ports will be pressed into service if the primary port to a segment fails for some reason.

4.3.3 Data Structures for Efficient Evaluation

A SmartBridge constructs certain data structures that enable efficient evaluation of the talker check and forwarding rule for each received packet. First, for each segment U connected to SmartBridge B, B determines if the undirected connection (B, U) appears in $NFST$. Connections that so appear are called *active* and the others *inactive*. Inactive connections can be ignored in the handling of network flood packets. Second, bridge B constructs a *network flood previous hop* table $NFPH_B$ which maps each segment S in the network to the unique segment T such that (T, B) appears in NFS_S. We write this as $NFPH_B[S] = T$. A network flood packet from S is supposed to reach B via T, which is to say that T is supposed to be the *previous hop* of a network flood packet from S arriving at B. $NFPH_B$ is easily computed using a single traversal of the $NFST$ starting at B.

SmartBridge B uses the value of $NFPH_B[S]$ to determine how its active connections are directed in NFS_S. Given any active connection (B, U), note that B is the network flood talker for S on U exactly when $U \neq NFPH_B[S]$. This information is what B uses to evaluate the talker check and forwarding rule for the network flood packet NFK_S.

4.3.4 Causal Ordering Among Multicast Packets

As will be described the next section, shortest path forwarding has a talker check and forwarding rule for $BPK_{S,D}$ based (in part) on a directed source tree BPS_S just like the check and rule for NFK_S based on NFS_S. It would be possible to ignore NFS_S and instead use BPS_S.

The reason to have a separate NFS_S based on $NFST$ is to provide causal ordering among multicast packets. That is, if (1) host A

[1] In practice, since we already construct a location revision spanning tree ($LRST$) to manage host location revision (§4.2), we just reuse the $LRST$ for the $NFST$.

sends multicast M1, (2) host B receives M1, and (3) host B subsequently sends multicast M2, then any host C that receives both M1 and M2 will receive M1 before M2. (This assumes bridges do not reorder packets.) The ordering occurs because both packets travel across the same spanning tree, namely $NFST$.

Bridging systems that use a single spanning tree for all forwarding provide causal ordering among all packets. This is not possible for SmartBridges.

4.4 Shortest Path Forwarding

A packet whose source and destination hosts have known locations is called a *shortest path packet*. The key feature that distinguishes SmartBridges from traditional, spanning tree bridges is that SmartBridges guarantee to forward each shortest path packet along a shortest path from its source to its destination. By shortest paths, we mean those that have the fewest number of hops in the network topology graph. Clearly, the shortest path need not be unique. We write $BPK_{S,D}$ to refer to an arbitrary shortest path packet whose source and destination hosts are known to be on segments S and D, respectively.

SmartBridges use a special set of shortest paths, called a *best paths set*. Given a network topology graph G, a best paths set PP satisfies the following three *best path properties*:

shortest path For each pair of vertices S and D in G, PP contains a path from S to D that is a shortest path in G.

source tree For each vertex S in G, the union of all paths in PP that start at S forms a directed tree whose source is S. We call this tree BPS_S, the *best path source tree* for S.

destination tree For each vertex D in G, the union of all paths in PP that end at D forms a directed tree whose sink is D. We call this tree BPD_D, the *best path destination tree* for D.

Figure 5 illustrates these properties. A path in PP is called a *best path*. For each pair of vertices S and D in G, the **shortest path** property guarantees that PP contains at least one path from S to D, and from either of the tree properties it follows that this path is unique. Let the unique best path from S to D be called $P_{S,D}$.

Next we explain how SmartBridges use PP (actually the best path source and destination trees) to forward shortest path packets and to detect and correct errors in the known location of the source host. In §5 we explain how to construct these trees given G.

4.4.1 Forwarding Rule for $BPK_{S,D}$

The *best path forwarding rule* can be stated as follows: a SmartBridge must forward $BPK_{S,D}$ along directed connections that appear in the intersection of BPS_S and BPD_D. Forwarding must occur along each connection in the proper direction. Supposing SmartBridge B receives this packet via segment T, it forwards it onto all segments U such that (T, B, U) appears in BPS_S and (T, B, U) appears in BPD_D. Because of the **destination tree** property, there will be at most one such segment U.

Note that $P_{S,D}$, the unique best path from S to D, appears in both BPS_S and BPD_D. Therefore, if $BPK_{S,D}$ actually starts at segment S, the SmartBridges will forward it along $P_{S,D}$ from S to D.

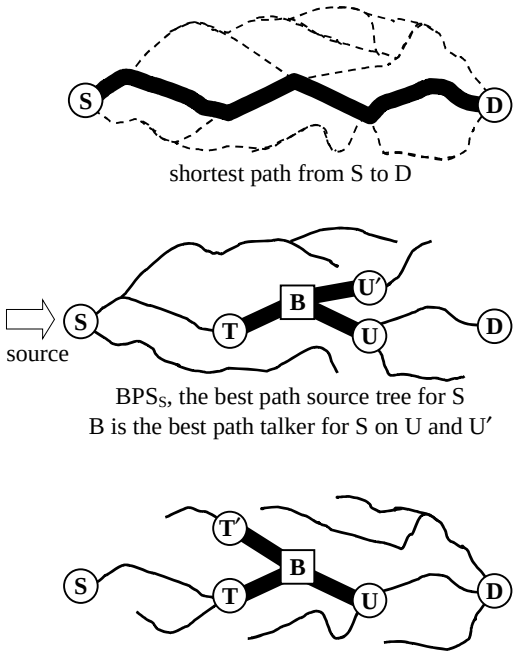

Figure 5: Best Path Properties.

4.4.2 Talker Detects a Moved Source Host

Note that any segment $U \neq S$ has a unique parent in BPS_S. We call this parent the *best path talker* for S on U. The situation is exactly analogous to the case of a network flood talker regarding a network flood packet (§4.3.2). Since no SmartBridge will forward $BPK_{S,D}$ except along directed connections that appear in BPS_S, the talker is the only bridge that could possibly forward $BPK_{S,D}$ onto U. The talker performs a *best path talker check* and triggers a host location revision accordingly. The same ability for a SmartBridge to distinguish its own packets is needed here as was discussed in the case of a network flood talker.

4.4.3 Data Structures for Efficient Evaluation

A SmartBridge constructs certain data structures that enable efficient evaluation of the talker check and forwarding rule for each received packet.

For each segment T connected to SmartBridge B, bridge B constructs a *best path next hop* table $BPNH_{T,B}$ which maps each destination segment D in the network to B's *next hop* on the best path from T to D, $P_{T,D}$. If $P_{T,D}$ does not go through B, then we say that B's next hop is $\perp$. We write an access to the table as $BPNH_{T,B}[D]$. Formally, $BPNH_{T,B}[D] = U$, where U is the segment such that (T, B, U) appears in BPD_D, if such a segment exists, otherwise $BPNH_{T,B}[D] = \perp$. Although at first glance it might appear that bridge B would have to examine the *destination tree* BPD_D for every D in order to compute $BPNH_{T,B}$, actually it suffices to perform a single traversal of the *source tree* BPS_T, as described later in §5.3.

Similarly, for every segment U connected to B, there is a *best path previous hop* table $BPPH_{U,B}$ which maps each source segment S to B's previous hop on $P_{S,U}$. $BPPH_{U,B}$ can be computed us-

ing a single traversal of the *destination tree* BPD_U. However, as described later in §5.1, our method of selecting a best paths set PP happens to produce symmetric paths, that is, $P_{S,D}$ is always the reverse of $P_{D,S}$. Consequently, $BPPH_{U,B}$ is identical to $BPNH_{U,B}$ and so it is not necessary to compute or store it separately.

Note that B is the best path talker for S on T exactly when $BPPH_{T,B}[S] \neq \perp$. When SmartBridge B receives a packet $BPK_{S,D}$ via segment T, it performs three table accesses to evaluate the talker check and forwarding rule:

1. If $BPPH_{T,B}[S] \neq \perp$ then trigger a location revision.
2. Let $U := BPNH_{T,B}[D]$. If $U = \perp$ then drop the packet.
3. If $T \neq BPPH_{U,B}[S]$ then drop the packet.

Otherwise SmartBridge B forwards $BPK_{S,D}$ to U.

5. COMPUTING BEST PATHS

The previous section described how SmartBridges operate assuming that a best paths set PP could be constructed. This section describes how to construct such a set.

5.1 Edge Weights

Our method for selecting a best paths set is based on an assignment of weights to edges in the network topology graph G. We say that a path from S to D is a *least-weight path* if there is no path from S to D of less total weight.

For any path P in G, the length of P, written $L(P)$, is the number of edges in P, and the weight of P, written $W(P)$, is the total of the weights of the edges in P. We assign weights to edges so that the following two *edge weight properties* are satisfied for each pair of vertices S and D in G:

shortest Any least-weight path from S to D is a shortest path in G.

unique The least-weight path from S to D is unique.

Given an edge weight assignment that satisfies these properties, it turns out that the set of least-weight paths is in fact a best paths set, as is easily proved by considering each of the best path properties in turn. The **shortest path** property is obvious. The **source tree** and **destination tree** properties follow from the facts that (1) least-weight paths are unique and (2) any subpath of a least-weight path must itself be a least-weight path.

Since a path has the same weight as its reverse, the best paths set selected by this edge weight scheme has a **symmetry** property: the reverse of any best path is itself a best path. This property causes the best path source tree for A and the best path destination tree for A to be identical—except that the direction of every connection is reversed—for any vertex A. This makes the implementation more efficient, since only one of these two trees and its derived data structures need to be computed.

We know of two ways of producing an assignment of edge weights that satisfy the edge weight properties: one based on a ranking of edges and the other based on a ranking of vertices. Our implementation uses the former scheme, although the latter would be slightly simpler to implement, as mentioned in the following section.

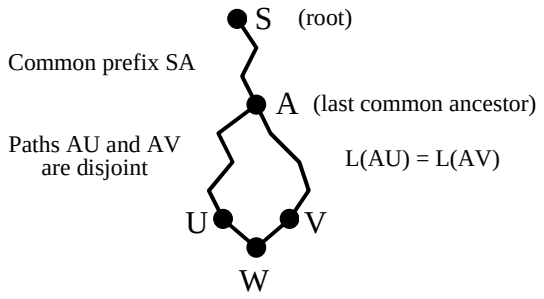

Figure 6: Breadth-First Discovery of Vertex W. *Vertex W has to choose between parent candidates U and V depending on the least weight path from the root S to W.*

Let each edge E be assigned a unique rank, written $r(E)$. (A rank is a positive integer: 1, 2, 3, etc.) Then we assign edge E the weight $1 + 2^{-r(E)}$. Since no edge can appear more than once in a shortest path, the edge weight properties follow from the use of distinct negative powers of two.

The alternative scheme using a ranking of vertices is similar. Let each vertex V be assigned a unique rank, written $r(V)$. Then we assign edge $E = (V_1, V_2)$ the weight $1 + 4^{-r(V_1)} + 4^{-r(V_2)}$. By a similar argument, it is easy to show that this assignment also satisfies the edge weight properties.

5.2 Best Path Traversal

Since least-weight paths are shortest paths, a breadth-first traversal of the network topology graph starting with S, which explores vertices in order of increasing minimum path length from S, also explores vertices along each least-weight path from S in order of increasing minimum path weight from S.

Recall that for any vertex S, the union of all best paths that start at S forms a spanning tree whose source is S, called the best path source tree for S, written BPS_S. So the set of best paths starting at S can be represented by recording in each vertex its parent in this tree.

When the traversal explores V, V's parent and the weight of the least-weight path from S to V are known and have been recorded in V. Visiting each of V's adjacent vertices W, the traversal computes the path weight from S via V to W and records it in W if it is the best so far. V is a *parent candidate* of W. Notice that W may have been previously visited from other parent candidates.

Figure 6 depicts the situation pictorially. Vertex W has to choose between parent-candidate vertices U and V based on the least-weight path from S. The least-weight paths from S to U and from S to V are known and can be retraced by backtracking through the vertices' parent fields.

An implementation could choose between parent candidates (e.g., U and V in Figure 6) by explicitly computing path weights, storing them with the parent field in each vertex, and comparing the path weights via each of the parent candidates. However, this *explicit*

path weight scheme has the disadvantage that path weights require a large number of bits—proportional to the number of edges or number of vertices in the graph, depending on whether edge ranking or vertex ranking is used—and thus is impractical for large-scale extended LANs.

It may be observed that the choice of best parent in the traversal depends only on the relative weights of the candidate paths, and not on the actual values of the weights. Hence, our implementation uses the following *implicit path weight* scheme which compares path weights without explicitly computing them.

Recall that when the traversal considers U and V as parent candidates of W, the best paths from S to U and from S to V are already known. These paths have some last common vertex, say A, possibly S itself. Thus, W's parent can be chosen by comparing the weights of two paths: from A to U to W and from A to V to W. Observe that the two paths have the same length, no common edges, and no common vertices except A and W. The edges (and vertices) on these two paths can be enumerated, in reverse, by starting at W, stepping back to U and V, and then simultaneously stepping back to their parents, grandparents, and so on, until a common ancestor is encountered, which will be A.

In the edge rank scheme, one path contains the edge of least rank among all edges on both paths and that path is guaranteed to have the larger weight. In the vertex rank scheme, one path contains the vertex of least rank among all vertices on both paths (disregarding A and W) and that path is guaranteed to have the larger weight.

The vertex rank scheme is slightly more efficient, because it does not require ranks to be stored with edges, thus simplifying the adjacency list representation of the graph somewhat.

5.3 Computing the BPNH Tables

Based on the network topology graph G and an edge weight assignment, each SmartBridge B computes a best path next hop table $BPNH_{T,B}$ for each segment T connected to B. Using a deterministic algorithm based on the same graph, each SmartBridge separately assigns the same edge weights, and therefore computes its $BPNH$ table entries based on the same best paths set.

After initializing all entries in $BPNH_{T,B}$ to $\perp$, SmartBridge B performs a best path traversal of G starting at T, as described in the previous section. When the traversal visits a vertex $W \neq T$, the parent V of W in the tree of best paths starting at T is known. Observe that if $V = B$ then W must be a segment connected to B and the best path from T to W must be (T, B, W). In this case, B's next hop on $P_{T,W}$ is W itself, so we set $BPNH_{T,B}[W] := W$. Otherwise W inherits the same next hop as its parent V, so we set $BPNH_{T,B}[W] := BPNH_{T,B}[V]$.

6. IMPLEMENTATION

We have implemented the SmartBridge design on two different types of inexpensive PC hardware. The first type is used to bridge 10 Mb/s Ethernets and the second is used to bridge 100 Mb/s Ethernets.

Our 10 Mb/s bridge consists of a 66 MHz Intel 486 processor and three identical Ethernet controllers. The Ethernet controllers plug into the ISA bus and use the 3Com 3C589 chip. This controller uses programmed I/O. The ISA bus is 16 bits wide and runs at 8 MHz.

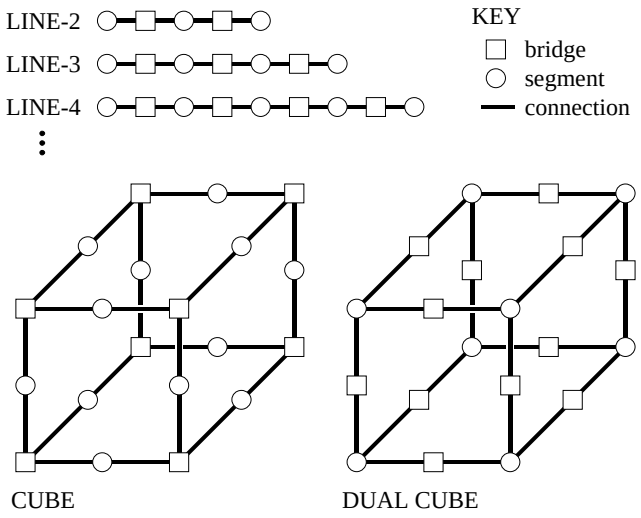

Figure 7: Reconfiguration Test Topologies. *Cube and Dual Cube are dual graphs of each other.*

Our 100 Mb/s bridge has a 400 MHz Celeron processor. It uses the Intel EtherExpress network controller, which plugs into the PCI bus. This controller supports DMA. The PCI bus is 32 bits wide and runs at 66 MHz.

The bridge boots a stock FreeBSD kernel (release 3.2) with the bridging code embedded in it as a completely separate module. We have not altered the Ethernet device driver in any way except to add a filter at the input routine to pass incoming packets to the bridging code.

The three primary performance metrics are (1) the time to do a reconfiguration, (2) the steady state throughput of a SmartBridge, and (3) the packet latency of going through a SmartBridge. Of these, reconfiguration times are highly dependent on the network topology.

It is important to keep reconfiguration short because normal packet forwarding is suspended during this time. Hosts sending packets during reconfiguration will notice dropped packets. Reliable transmission protocols like TCP will timeout if packets are dropped for a sufficiently long time [5].

6.1 Reconfiguration Time

We measured the reconfiguration time of our 10 Mb/s system in each of the topologies shown in Figure 7 and Figure 8 plots the results. Since the clock granularity in our system is 10 milliseconds, times less than one clock tick were measured by averaging a large number of runs. Observe that the reconfiguration time for the *line* topology increases almost linearly as bridges are added. This is not surprising, since the diameter of the network (and thus the diameter of the propagation order spanning tree) increases linearly with the number of bridges. In fact, for a given number of bridges and ports, the line topology gives the worst case reconfiguration time, but with twelve bridges the reconfiguration time is still only about 16 milliseconds. This is well less than the timeout period of protocols like TCP, which is typically on the order of 500 milliseconds.

Although we extrapolate that our system could scale to a line topology containing about two hundred bridges, any realistic network

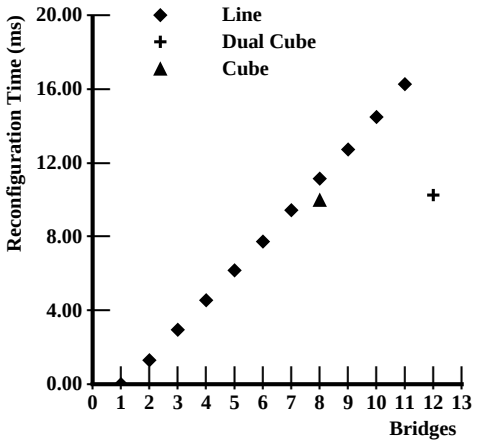

Figure 8: Reconfiguration Times. *Time to reconfigure SmartBridges in various topologies is plotted against the number of bridges.*

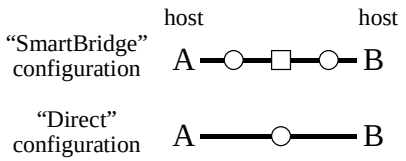

Figure 9: Throughput Test Configurations.

would have many more interconnections and thus a smaller network diameter for a given number of bridges. The cube and dual cube topologies are examples of such more realistic networks, and their reconfiguration times can be seen to be less than that of the line topology for the same number of bridges. Link speeds faster than 10 Mb/s and faster CPUs in bridges would also shorten the reconfiguration time and allow even larger configurations.

Our reconfiguration time does not include the time to detect and stabilize an inventory change; that takes an additional 15 ms. The detection time depends on the granularity of the FreeBSD clock and could be shortened by making the clock tick faster.

6.2 Throughput and Latency

We used *netperf* [11] to measure the maximum throughput from one host to another at various packet sizes both with and without a SmartBridge between them. Link and transport overheads are not counted in the throughput. If these overheads are included, our 100 Mb/s system runs at full link bandwidth for maximum-size packets.

Figure 9 shows the two configurations. The "SmartBridge" configuration has two hosts on different segments connected by a SmartBridge. The "Direct" configuration has two hosts directly connected to the same segment.

Figure 10 plots the throughput of our 10 Mb/s system. The SmartBridge is capable of sustaining 8.4 Mb/s with large packets at a CPU utilization of over 90%. Without a SmartBridge, directly connected hosts can achieve a throughput of 9.6 Mb/s. The principal reason that the SmartBridge does not saturate the link is that the network controller uses programmed I/O to move data to and from the network, and the CPU saturates before the link saturates.

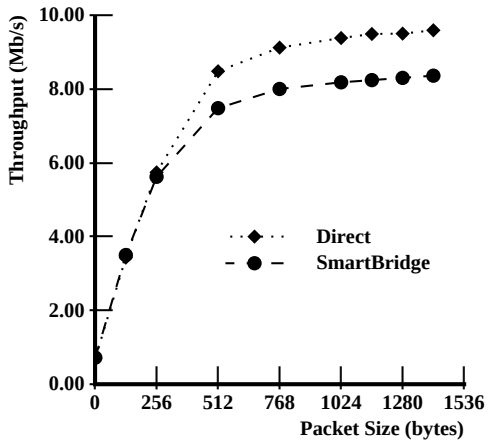

Figure 10: Throughput of the 10 Mb/s system.

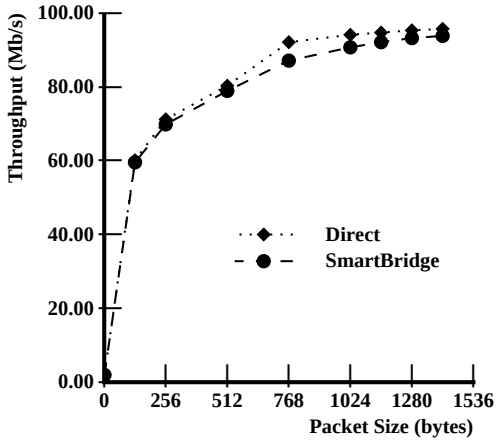

Figure 11: Throughput of the 100 Mb/s system.

With a controller capable of doing DMA, the throughput of the SmartBridge is much better. Figure 11 plots the throughput of our 100 Mb/s system. In this case, SmartBridge throughput is very close to direct throughput. CPU utilization is only 35% at large packet sizes.

We also used *netperf* to measure the additional latency due to a SmartBridge. The additional latency was almost exactly one store-and-forward delay. This is not surprising, given that our commodity network controllers do not permit cut-through.

6.3 Table Sizes

Each SmartBridge stores information in several tables. The sizes of these tables depend on the number of bridges, b, the number of segments, s, the average number of ports on a bridge, p, the number of hosts, h, and the average number of bridge ports connected to an segment, i. Note that $i = b * p/s$.

In our implementation, the maximum of vertices is 2048, the maximum number of ports per bridge is 128, and the maximum number of hosts allowed is 8192. Thus $(b + s) \leq 2048$, $p \leq 128$ and $h \leq 8192$.

With the data representation in our implementation, the network topology graph and related working space takes about $8 * (b +$

s) bytes for vertices and about $5 * (b * p)$ bytes for edges. A given SmartBridge B computes and stores only the relevant network flood and best path tables: the network flood previous hop table $NFPH_B$ takes $8 * s$ bytes and the best path next hop tables $BPNH_{*,B}$ take about $7*(p*s)$ bytes total. The host location table takes $12*h$ bytes, and the inventory construction process consumes about $20 * (i * p)$ bytes in addition to debugging data. This figure can be further lowered if we use short integers instead of linked list pointers.

6.4 Comparison with Spanning Tree Bridges

A spanning tree bridge stores a host direction table that is approximately the same size as a SmartBridge's host location table. In a typical network, the number of hosts should be 10 to 100 times the number of bridges and segments, so we expect the host location table to consume the majority of available storage in each SmartBridge. Hence the storage requirement is approximately the same between SmartBridges and spanning tree bridges.

In both bridge designs, access to the host table requires an associative lookup based on host UID, and such an access is performed for both the source and destination addresses of each packet. A SmartBridge also performs accesses to the $NFRT$ or the $BPFT$ on each packet, but the implementation can arrange to represent segments by small integer indexes, thus making the $NFRT$ or $BPFT$ accesses quite efficient. Hence the amount of processing required to forward a packet is approximately the same between SmartBridges and spanning tree bridges.

A spanning tree bridge system requires an enormous amount of time to reconfigure after a topology change: on the order of 30 to 90 seconds. This results from the fact that the spanning tree protocol can move through inconsistent states, which could create forwarding loops; to avoid such loops the bridges suspend forwarding for a length of time guaranteed to allow the protocol to converge. A SmartBridge system reconfigures in about 20 milliseconds—three orders of magnitude faster—because it uses a diffusing computation to obtain control of the network and move efficiently to completion.

6.5 Shortcomings

Our current design and implementation has some shortcomings, which could be overcome by further work.

We treat all LANs as equally good for forwarding traffic. It would be better to prefer a higher-speed LAN over a lower-speed LAN, even to the extent of preferring several high-speed hops to one low-speed hop.

We treat all traffic as equally deserving of service. It would be better to regulate broadcast traffic (by dropping excessive broadcasts) so that a broadcast storm could not deny service to unicast traffic.

We treat anyone sending SmartBridge protocol messages as if it were a SmartBridge. It would be better to use cryptographic authentication to verify that the sender really should be trusted.

7. RELATED WORK

Apart from the SmartBridge design, there are three general approaches to mitigating the congestion problem in spanning tree bridges that are currently in use: replication, crossover, and tunneling. However, none of these techniques guarantees to send packets on the shortest path between hosts.

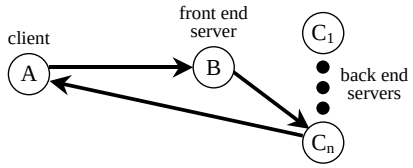

Figure 12: Triangle routing. *Client A sends a request to a front-end server B that forwards it to one of several back-end servers C.*

7.1 Replication

One approach to mitigate the congestion problem in spanning tree bridges is to create not one, but several spanning trees, each with a different root. This is most easily done by creating different virtual universes. Each universe would have its own spanning tree, with a different root and set of active connections. A hash function over source and destination is applied to each packet to determine the universe to which the packet belongs, and the packet is forwarded using the spanning tree for that universe.

Although this approach tends to spread the load across different root bridges and different redundant connections, there is nothing to prevent two hosts from getting unlucky and landing in a universe with a bad spanning tree from their perspective. A more serious weakness of this approach is that the location of each host must be learned separately in each universe. This can be a serious problem for triangle-routing protocols such as illustrated in Figure 12, where the location of a packet's destination may never be learned in the universe to which that packet belongs, since learning is based on a packet's source.

The multi-universe approach can easily be applied to SmartBridge, with a different set of best paths in each universe. The SmartBridge system would still learn the segment on which a host is located with the first packet sent by that host, but, unlike spanning tree bridges, this knowledge can be used in all universes.

7.2 Crossover

Under special circumstances, a segment that is not part of the spanning tree can be used to divert load from the root. Imagine bridges A and B that are connected by a redundant segment. Using the standard spanning tree learning process, A learns about all the hosts located below it in the spanning tree, as does B. Then, using a special protocol, A and B can exchange their information about their hosts and agree to use the redundant segment to forward packets directly to the other bridge for hosts that are below it in the spanning tree. This scheme and extensions are described more fully by Perlman [16]. Unlike SmartBridges, crossover schemes are ad-hoc approaches to the problems of load balancing. They place fairly tight topological constraints and do not necessarily improve the performance of all hosts.

7.3 Tunneling

Tunneling is a scheme where multiple LANs (or extended LANs) are connected by "tunnels", which are dedicated paths between pairs of LANs (or extended LANs). Tunnels are created by wrapping LAN packets within a larger data frame and using any convenient network protocol to carry the encapsulated packets between the tunnel's end points. The standard LAN (or bridge) protocol is used to carry traffic within each LAN (or extended LAN).

VLAN [10] is similar to tunneling.

Tunnels have some attractive features. They can keep extended LANs (and the corresponding spanning trees) from getting very large. They also offer the flexibility of routing traffic from specific LANs, or even better, only packets belonging to specific protocols. Unfortunately, tunneling requires manual configuration and set up because tunnel end points have to know about each other. Tunnels are complementary to SmartBridges in that extended LANs connected by SmartBridges can be tunneled.

8. CONCLUSIONS

The problem of alleviating congestion in extended LANs is long-standing, and SmartBridge is the first comprehensive solution that we are aware of.

SmartBridges route packets using shortest paths and require little or no human administration or configuration. They can be built inexpensively using standard, off-the-shelf hardware. The algorithms at the heart of the design are simple to implement and give good performance.

Although methods for performing incremental updates to the topology will be needed at extremely large scales, our "global reboot" approach is simple, easily proven correct, and fast enough to handle large scale extended LANs involving dozens to hundreds of bridges.

As 100 Mb/s switched Ethernet and Gigabit Ethernet become more common, the number of bridges in an extended LAN is likely to increase, making the SmartBridge solution even more valuable.

Our future plans include testing the design on higher speed networks, validating its scalability in larger configurations and incorporating it into switches. As a practical matter, SmartBridges must also cooperate with spanning tree bridges in the same network, and we have a preliminary solution, which we plan to test further.

Acknowledgements

We would like to thank our colleagues Mark Lillibridge and Raymie Stata for many illuminating technical discussions. Thanks are also due to our anonymous referees and our shepherd Radia Perlman.

9. REFERENCES

[1] Yehuda Afek, Baruch Awerbuch, and Eli Gafni. Applying static network protocols to dynamic networks. In *28th Annual Symposium on Foundations of Computer Science*. IEEE, October 1987.

[2] Baruch Awerbuch, B. Patt, and George Varghese. Self-stabilization by local checking and correction. In *32nd Annual Symposium on Foundations of Computer Science*, pages 268–277. IEEE, October 1991.

[3] Kenneth P. Birman. *Building Secure and Reliable Network Applications*, chapter 13.10: The Group Membership Problem, pages 253–265. Manning, 1996.

[4] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, pages 47–76, February 1987.

[5] Douglas E. Comer and David L. Stevens. *Internetworking with TCP/IP Volume II*. Prentice Hall, 1991.

[6] Digital Equipment Corporation. *DIGITAL Standard 200-1 DIGITAL Network Architecture—Maintenance Operations Functional Specification, Version T4.0.0*, B1 edition, January 1988. Document number A-DS-EL00200-01-0000, Compaq Computer Corporation, Houston, Texas.

[7] Edsger W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, August 1980.

[8] IBM. The NETBIOS frames protocol. In *IBM Local Area Network Technical Reference*, December 1990. Document number SC30-3383-03.

[9] IEEE. *Local and Metropolitan Area Networks: Common Specifications: Media Access Control (MAC) Bridges*. ANSI/IEEE Std 802.1D-1998.

[10] IEEE. *Local and Metropolitan Area Networks: Virtual Bridge Local Area Networks*. IEEE Std 802.1Q-1998.

[11] Rick Jones. *The Public NetPerf Homepage*. available at http://www.netperf.org/netperf/NetperfPage.html.

[12] David E. McDysan and Darren L. Spohn. *ATM Theory and Applications*. McGraw-Hill, 1999.

[13] J. M McQuillan, I. Richer, and E. C. Rosen. The new routing algorithm for the ARPANET. *IEEE Transactions on Communications*, 28(5):711–719, May 1980.

[14] David Oran. *OSI IS–IS Intra-domain Routing Protocol*. RFC 1142. Available from http://www.rfc-editor.org/rfc.html. A republication of ISO DL 10589.

[15] Radia Perlman. An algorithm for distributed computation of a spanning tree in an extended LAN. In *Proceedings of the Ninth Data Communications Symposium*, pages 44–53. ACM, 1985.

[16] Radia Perlman. *Interconnections: Bridges and Routers*, chapter 3.7.2: Using Extra Links, pages 86–94. Addison-Wesley, 1992. 1st edition.

[17] Thomas L. Rodeheffer and Michael D. Schroeder. Automatic reconfiguration in Autonet. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, pages 183–197, 1991.

[18] Michael D. Schroeder et al. Autonet: A high-speed, self-configuring local area network using point-to-point links. *IEEE Journal on Selected Areas in Communications*, October 1991.