EtherFuse: An Ethernet Watchdog

Khaled Elmeleegy, Alan L. Cox, T. S. Eugene Ng
Department of Computer Science
Rice University

Abstract

Ethernet is pervasive. This is due in part to its ease of use. Equipment can be added to an Ethernet network with little or no manual configuration. Furthermore, Ethernet is self-healing in the event of equipment failure or removal. However, there are scenarios where a local event can lead to network-wide packet loss and congestion due to slow or faulty reconfiguration of the spanning tree. Moreover, in some cases the packet loss and congestion may persist indefinitely.

To address these problems, we introduce the EtherFuse, a new device that can be inserted into an existing Ethernet to speed the reconfiguration of the spanning tree and prevent congestion due to packet duplication. EtherFuse is backward compatible and requires no change to the existing hardware, software, or protocols. We describe a prototype EtherFuse implementation and experimentally demonstrate its effectiveness. Specifically, we characterize how quickly it responds to network failures, its ability to reduce packet loss and duplication, and its benefits on the end-to-end performance of common applications.

Categories and Subject Descriptors

C.2.1 [**Network Architecture and Design**]: Packet Switching Networks; C.2.3 [**Network Operations**]: Network Monitoring; C.2.5 [**Local and Wide-Area Networks**]: Ethernet

General Terms

Management, Performance, Reliability, Experimentation

Keywords

Network Watchdog, Ethernet, Reliability, Count to Infinity, Forwarding Loop

1. INTRODUCTION

This paper introduces the EtherFuse, a new device that can be inserted into an existing Ethernet in order to increase the network's robustness. The EtherFuse is backward compatible and requires no change to the existing hardware, software, or protocols.

SIGCOMM'07, August 27–31, 2007, Kyoto, Japan.

Ethernet is the dominant networking technology in a wide range of environments, including home and office networks, data center networks, and campus networks. Moreover, Ethernet is increasingly used in mission-critical applications. Consequently, the networks supporting these applications are designed with redundant connectivity to handle failures.

Although modern Ethernet is based on a point-to-point switched network technology, Ethernet still relies on packet flooding to deliver a packet to a new destination address whose topological location in the network is unknown. Moreover, Ethernet relies on switches observing the flooding of a packet to learn the topological location of an address. Specifically, a switch observes the port at which a packet from a particular source address S arrives. This port then becomes the outgoing port for packets destined for S and so flooding is not required to deliver future packets to S.

To support the flooding of packets for new destinations and address learning, an Ethernet network dynamically builds a cycle-free active forwarding topology using a spanning tree protocol. This active forwarding topology is a logical overlay on the underlying physical topology. This logical overlay is implemented by configuring the switch ports to either forward or block the flow of data packets.

Redundant connectivity in the physical topology provides protection in the event of a link or switch failure. However, it is essential that the active forwarding topology be cycle free. First of all, broadcast packets will persist indefinitely in a network cycle because Ethernet packets do not include a time-to-live field. Moreover, unicast packets may be mis-forwarded if a cycle exists. Specifically, address learning may not function correctly because a switch may receive packets from a source on multiple switch ports, making it impossible to build the forwarding table correctly.

The dependability of Ethernet therefore heavily relies on the ability of the spanning tree protocol to quickly recompute a cycle-free active forwarding topology upon a network failure. While the active forwarding topology is being recomputed, localized packet loss is to be expected. Unfortunately, under each of the standard spanning tree protocols [16, 15], there are scenarios in which a localized network failure can lead to network-wide packet loss and congestion due to slow or faulty reconfiguration of the spanning tree or the formation of a forwarding loop.

Detecting and debugging the causes of these problems is labor intensive. For example, Cisco's prescribed way for troubleshooting forwarding loops is to maintain a current diagram of the network topology showing the ports that block data packets. Then, the administrator must check the state of each of these ports. If any of these ports is forwarding, then a failure at that port is the likely cause for the forwarding loop [8]. However, when there is network-wide packet loss and congestion, remote management tools may

also be affected, making it difficult to obtain an up-to-date view of the network state. Consequently, the network administrator may have to walk to every switch to check its state, which can be time consuming.

The network disruption at the Beth Israel Deaconess Medical Center in Boston illustrates the difficulty of troubleshooting Ethernet failures [2, 3]. In this incident, the network suffered from disruptions for more than three days due to problems with the spanning tree protocol.

A variety of approaches have been proposed to address the reliability problems with Ethernet. Some researchers have argued that Ethernet should be redesigned from the ground up [19, 18, 20]. In contrast, others have proposed keeping the basic spanning tree model but changing the protocol responsible for its maintenance to improve performance and reliability [11]. Proprietary solutions to a few specific spanning tree problems have been implemented in some existing switches, including Cisco's Loop Guard [6] and Unidirectional Link Detection (UDLD) protocol [9]. However, together these proprietary solutions still do not address all of Ethernet's reliability problems.

Instead of changing the spanning tree protocol, we introduce the EtherFuse, a new device that can be inserted into redundant links in an existing Ethernet to speed the reconfiguration of the spanning tree and prevent congestion due to packet duplication. The EtherFuse is compatible with any of Ethernet's standard spanning tree protocols and requires no change to the existing hardware, software, or protocols. In effect, the EtherFuse allows for the redundant connectivity that is required by mission critical applications while mitigating the potential problems that might arise.

We describe a prototype EtherFuse implementation and experimentally demonstrate its effectiveness. Specifically, we characterize how quickly it responds to network failures, its ability to reduce packet loss and duplication, and its benefits on the end-to-end performance of common applications.

The rest of this paper is organized as follows. The next section describes the problems that are addressed by the EtherFuse. Section 3 describes the EtherFuse's design and operation. Section 4 describes a prototype implementation of the EtherFuse. Section 5 describes our experimental setup. Section 6 presents an evaluation of the EtherFuse's effectiveness. Section 7 discusses related work. Finally, Section 8 states our conclusions.

2. ETHERNET FAILURES

The three IEEE standard Ethernet spanning tree protocols are the Spanning Tree Protocol (STP), the Rapid Spanning Tree Protocol (RSTP), and the Multiple Spanning Tree Protocol (MSTP). RSTP was introduced in the IEEE 802.1w standard and revised in the IEEE 802.1D (2004) standard. It is the successor to STP. It was created to overcome STP's long convergence time that could reach up to 50 seconds [7]. In STP, each bridge maintains a single spanning tree path. There are no backup paths. In contrast, in RSTP, bridges compute alternate spanning tree paths using redundant links that are not included in the active forwarding topology. These alternate paths are used for fast failover when the primary spanning tree path fails. Moreover, to eliminate the long delay used in STP for ensuring the convergence of bridges' spanning tree topology state, RSTP bridges use a hop-by-hop hand-shake mechanism called *sync* to explicitly synchronize the state among bridges. A tutorial by Cisco [10] provides a more detailed description of RSTP. MSTP is defined by the IEEE 802.1Q-2003 standard. It was created to support load balancing within networks having multiple VLANs. Specifically, in contrast to RSTP and STP, MSTP allows for the creation of multiple spanning trees within the network and the assignment of one or more VLANs to each of these spanning trees. Many of its basic mechanisms are derived from RSTP.

The remainder of this section describes the problems that can occur under each of these spanning tree protocols that are addressed by the EtherFuse.

2.1 Count to Infinity

The RSTP and MSTP protocols are known to exhibit count-to-infinity behavior under some failure conditions [18, 11]. Specifically, count to infinity can occur when the network is partitioned and the root bridge of a spanning tree is separated from one or more physical cycles. During count to infinity, the spanning tree topology is continuously being reconfigured and ports in the network can oscillate between forwarding and blocking data packets. Consequently, many data packets may get dropped.

To construct a spanning tree, every bridge in the network has a port that connects it to its parent in the tree. This port is called the root port and it is used for connectivity to the root bridge. In RSTP and MSTP, some bridges have ports with a different role, called alternate ports. An alternate port exists if there is a loop in the physical topology, and it is blocked to cut this loop. An alternate port also caches topology information about an alternate path to the root bridge which gets used if the bridge loses connectivity to the root bridge through its root port.

When the root bridge is separated from a physical cycle, the topology information cached at an alternate port in the cycle immediately becomes stale. Unfortunately, RSTP and MSTP use this stale topology information and spread it through protocol messages called Bridge Protocol Data Units (BPDUs), triggering a count to infinity. The count to infinity ends when the message age of the BPDUs carrying stale information reaches a limit. A smaller value of this limit would decrease the duration of a count to infinity. However, because a BPDU's message age increases as it passes from one bridge to the next, this limit also imposes an upper bound on the height of the spanning tree. Thus, having a small value for this limit restricts the size of the overall network. Another factor that increases the duration of the count to infinity is that RSTP specifies that a bridge can only transmit a limited number of BPDUs per port per second. This number is given by the `TxHoldCount` parameter. RSTP uses this parameter to limit processing load. Altogether, these factors can make a count to infinity last for tens of seconds [11].

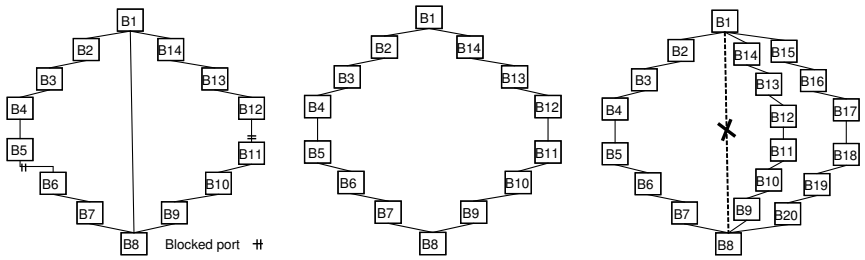
2.2 Forwarding Loops

Having forwarding loops in Ethernet can be disastrous. A forwarding loop can cause packets to keep circulating inside the network. Also, if broadcast packets get trapped in a loop, they will generate broadcast storms. Moreover, multiple forwarding loops can cause the trapped packets to multiply exponentially.

At a high level, a forwarding loop is formed when a bridge's port erroneously switches from a blocked state to a forwarding state where it starts forwarding data packets. Forwarding loops can be short lived allowing the network to quickly resume normal operation after the loop is broken, or they can be long lived or even permanent rendering the network unusable. In the following, we explain various cases where forwarding loops can form under each of the spanning tree protocols. Finally, we explain how forwarding loops can break Ethernet's address learning mechanism.

2.2.1 BPDU Loss Induced Forwarding Loops

In all of the spanning tree protocols a port is blocked if it is not the root port and it receives BPDUs from a peer bridge that advertise a lower-cost path to the root bridge than the BPDUs it sends.



(a) Before link failure. (b) After failure. (c) Multiple loops.

Figure 1: Examples of permanent forwarding loops forming under STP when the spanning tree height exceeds the `MaxAge` limit due to a link failure. In the figure, only blocked ports are not forwarding data packets. The network has the `MaxAge` value set to 6.

However, if the port fails to receive BPDUs from its peer bridge for an extended period of time, it may start forwarding data. For example, in RSTP, this period is three times the interval between regular BPDU transmissions. BPDU loss can be due to an overload of some resource, like the control CPU or a link's bandwidth. It can also be because of hardware failures and bugs in the firmware running on Ethernet bridges.

One scenario in which the control CPU can become overloaded is when a bridge's CPU is involved in the processing of data packets. Normally, Ethernet frames circulating around a forwarding loop are handled by the bridge's line cards in hardware and would not be processed by the control CPU. However, sometimes bridges do Internet Group Management Protocol (IGMP) snooping to optimize IP multicast [5]. Since IGMP packets are indistinguishable from multicast data packets at layer 2, a switch running IGMP snooping must examine every multicast data packet to see whether it contains any pertinent IGMP control information. If IGMP snooping is implemented in software, the switch's CPU can get overwhelmed by multicast packets. When the control CPU is overloaded, it may no longer process and transmit BPDUs in a timely fashion.

A subtle case of a BPDU loss induced forwarding loop can result from a uni-directional link. Although Ethernet links are normally bi-directional, the failure of a transceiver on an optical fiber link can cause the link to become uni-directional. In this case, if BPDUs are transmitted in the failed direction, they will be lost. Such BPDU loss can cause a port that was previously blocked to suddenly start forwarding data packets in the direction that is still functional and create a forwarding loop. Thus, a single transceiver failure can lead to a permanent forwarding loop.

2.2.2 `MaxAge` *Induced Forwarding Loops*

In all the Ethernet spanning tree protocols, the maximum height of the spanning tree is limited. In STP and RSTP, the limit is given by the `MaxAge`. Whereas, in MSTP, the limit is given by the `TTL` in the BPDUs from the root bridge. If a network is too large, the BPDUs from the root bridge will not reach all bridges in the network. Suppose bridge A sends a BPDU to bridge B but the BPDU arrives with a message age equal to `MaxAge` or a `TTL` equal to zero. Under RSTP and MSTP, B would block its port to A, partitioning the network. However, under STP the BPDU with the maximum message age is completely ignored by B. The end result is as if B is not connected to A at all, and the port connecting B to A will become forwarding by default. Moreover, those distant bridges that do not receive BPDUs from the true root bridge will try to construct a spanning tree among themselves. The two spanning trees in the network can be conjoined at the leaves and lead to a forwarding loop.

Unfortunately, such a forwarding loop can form as a result of a single, localized failure. For example, a single link failure can cause a perfectly functional Ethernet network to violate the maximum spanning tree height limit, leading to a sudden, complete network breakdown. Figure 1 gives an example of such a problem. B1 is the root bridge of the network. Assume that the value of `MaxAge` is set to 6. Figure 1(a) shows the network before the failure, and Figure 1(b) shows what happens after the link connecting bridge B1 to bridge B8 fails.

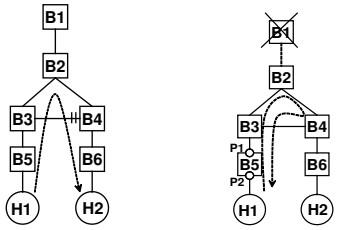
Before the failure, all bridges are within 4 hops of the root bridge B1. The blocked ports at B5 and B11 cut the physical cycles to create a spanning tree. However, after the failure, B8 becomes 7 hops away from B1. As a result, the message age of BPDUs from B1 will have reached `MaxAge` when they arrive at bridge B8 and so they are dropped by B8. Without receiving any valid BPDUs from its neighbors, bridge B8 believes that it is the root of the spanning tree and makes the ports that connect it to bridges B7 and B9 designated ports. On the other hand, both B7 and B9 believe that B1 is the root of the spanning tree as the BPDUs they receive conveying this information have message age below `MaxAge`. B7 and B9 both believe that B8 is their child in the spanning tree and thus they make their ports connecting them to B8 designated ports. All the ports in the network are thus forwarding data packets, and a permanent forwarding loop is formed.

In networks with more complex topologies, this problem can create multiple forwarding loops. Figure 1(c) generalizes the previous example to illustrate the formation of multiple forwarding loops after the failure of the link between bridges B1 and B8. In this network, broadcast packets will be replicated at the junctions at B1 and B8. This creates an exponentially increasing number of duplicate packets in the network that may render the entire network inoperative.

2.2.3 *Count to Infinity Induced Forwarding Loops*

All of the spanning tree protocols are specified as a set of concurrent state machines. It has been discovered that in RSTP the combination of the count-to-infinity behavior, a race condition between RSTP state machines, and the non-determinism within a state machine can cause an Ethernet network to have a temporary forwarding loop during the count to infinity. This problem is explained in detail in [12]. Here, we provide a sketch of the explanation.

Normally, during a count to infinity, a hand-shake operation between adjacent bridges, called *sync*, prevents a forwarding loop from forming. However, a race condition between two RSTP state machines and a non-deterministic transition within a state machine that together allow a sync operation to be mistakenly bypassed. Once the sync operation is bypassed, a forwarding loop is formed which lasts until the end of the count to infinity.



(a) Before failure. (b) After failure of the root bridge.

Figure 2: Forwarding table pollution caused by a temporary forwarding loop.

2.2.4 Pollution of Forwarding Tables

Forwarding tables in Ethernet are learned automatically. When a bridge receives a packet with a source address A via a port p, p automatically becomes the output port for packets destined for A. This technique works fine in a loop-free topology. However, when a forwarding loop occurs, a packet may arrive at a bridge multiple times via different ports in the loop. This can cause a bridge to use the wrong output port for a destination address. Moreover, the effects of such forwarding table pollution can be long lasting.

Figure 2 shows an example of how forwarding tables can get polluted. Figure 2(a) shows the forwarding path, B5-B3-B2-B4-B6, between end hosts H1 and H2 in the absence of failure. The death of the root bridge, B1, can lead to a temporary forwarding loop among B2, B3 and B4 as explained in Section 2.2.3. Figure 2(b) shows how the forwarding table of bridge B5 can get polluted in the presence of a forwarding loop among B2, B3 and B4. Initially B5 believes H1 is connected to port P2. However after the forwarding loop is formed, a packet from H1 can reach B3 then spin around the loop to reach B3 again, which can send a copy back to B5[1]. Thus, bridge B5 receives a packet with source address H1 via port P1 and believes that port P1 should be the output port for H1. Once this mistake is made by B5, there is no way for H2's data packets to reach H1 even after the temporary forwarding loop has ended because those packets will be dropped by B5 as they arrive on port P1. This problem will only get fixed when the incorrect forwarding table entry at B5 times out, or when H1 transmits a data packet.

3. THE DESIGN OF THE ETHERFUSE

The EtherFuse is a device that can be inserted into the physical cycles in the network to improve Ethernet's reliability. It has two ports and is analogous to an electric circuit fuse. If it detects the formation of a forwarding loop, it breaks the loop by logically disconnecting a link on this loop. The EtherFuse can also help mitigate the effects of the count to infinity in RSTP and MSTP.

3.1 Detecting Count to Infinity

Count to infinity occurs around physical loops in Ethernet. The way that the EtherFuse detects a count to infinity is by intercepting all BPDUs flowing through it and checking if there are 3 BPDUs announcing an increasing cost to the same root R. The EtherFuse maintains a counter that is incremented every time the EtherFuse receives a BPDU with increasing cost to the same root. The counter

[1] The reason B3 may send a copy to B5 is either because this is a packet with a broadcast destination, or B3 does not have an entry for the destination of the packet in its forwarding table and thus it falls back to flooding the packet on all its ports. B3 may not have an entry for the packet's destination in its forwarding table either because this is the first time it hears of this destination, or because its forwarding table entry has timed out or has been flushed due to the reception of a BPDU instructing it to do so.

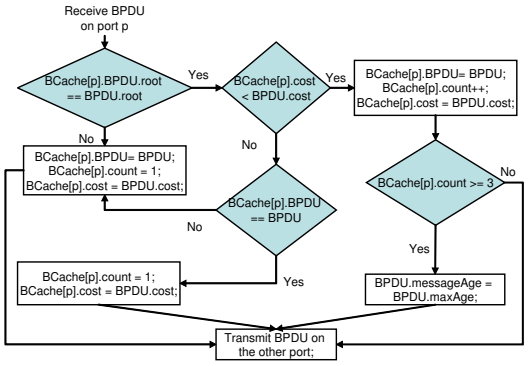


Figure 3: Flow chart of how the EtherFuse detects and mitigates a count to infinity.

is reset to one if the EtherFuse receives two consecutive identical BPDUs. If this counter reaches the value of 3, it signals that a count to infinity is taking place. This means that there is stale information about R that is circling around the loop and will keep doing so until it is aged out. The reason for checking for 3 consecutive BPDUs announcing increasing costs is that BPDUs are sent out if the bridge has new information to announce, or periodically every hello time, which is typically 2 seconds. Thus, it is unlikely that a path cost to the root will increase twice during two consecutive hello times, due to any reason other than a count to infinity. In the unlikely event that the there was no count to infinity but the network was reconfigured twice during two consecutive hello times, the BPDU following the two BPDUs with increasing costs will announce the same cost as the preceding one. Thus, the EtherFuse will realize that no count to infinity is taking place and it will not take any further action, leaving the network to resume its normal operation.

The EtherFuse does the BPDU monitoring independently for each of its 2 ports. It uses a BPDU cache (BCache) that maintains the state of BPDUs it has received at each port. Figure 3 shows a flow chart explaining how the EtherFuse detects a count to infinity. Since fresh information can chase stale information around the loop announcing two different roots during a count to infinity, the cache has two entries per port to record both the fresh and the stale information. Only two entries are used in the cache because during the count to infinity there can be BPDUs announcing at most two different roots [11]. Both the fresh and the stale information are cached because the EtherFuse can not distinguish between them. Thus, it monitors both copies in the cache checking if either of them exhibit two consecutive increases in cost. The details about maintaining two cache entries per port in the BCache are omitted from Figure 3 for simplicity.

3.2 Detecting Forwarding Loops

The key idea for detecting forwarding loops in Ethernet is by detecting packets that are circling around the loop. The EtherFuse takes a hybrid approach of passively monitoring traffic in the network to infer the existence of a forwarding loop, and actively probing the network to verify the loop's existence. Passive monitoring is preferred as it does not introduce extra network traffic. Moreover, because passive forwarding loop detection takes advantage of the data packets flowing through the network, it is likely to be faster than any practical method based on periodic active probing.

To monitor the network for forwarding loops, EtherFuse checks for duplicate packets. This is because if there is a forwarding loop, a packet may spin around the loop and arrive again at the EtherFuse. The EtherFuse checks for duplicates by keeping a history of the

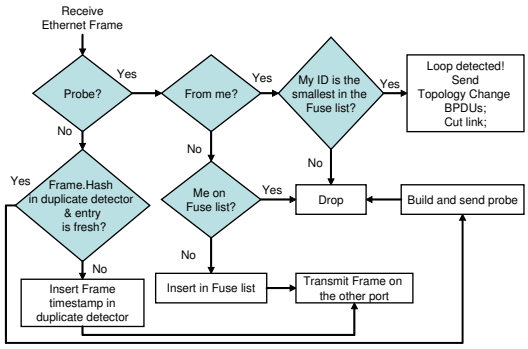


Figure 4: Flow chart of how the EtherFuse detects and stops forwarding loops.

hashes of the packets it received recently. Every new incoming packet's hash is checked against this history. If a fresh copy of the packet's hash is found, then the packet is a duplicate signaling a potential forwarding loop. A hash in the history is fresh if its timestamp is less than the current time by no more than a threshold. This threshold should be no less than the maximum network round trip time. Otherwise, a packet's hash may expire before the packet completes a cycle around the loop. If no fresh copy of the received packet's hash is found in the history, the hash is recorded in the history along with its timestamp. As an optimization, the Ethernet frame's Cyclic Redundancy Check (CRC) can be used as the hash of the packet's contents.

By itself, this forwarding loop detection technique may have false positives due to collisions between hashes of different packets or a malicious end host intentionally injecting duplicate packets into the network to trick the EtherFuse into thinking that there is a forwarding loop. To avoid false positives in such cases, the EtherFuse uses explicit probing once it suspects the existence of a forwarding loop. These probes are sent as Ethernet broadcast frames to guarantee that if there is a loop they will go around it and not be affected by forwarding tables at the Ethernet switches. The source address of the probe is the EtherFuse's MAC address. If the EtherFuse receives a probe it has sent then this implies that there is a forwarding loop. However, the probe may get dropped even in the presence of a forwarding loop. In this case, the fuse will receive more duplicate packets forcing it to send more probes until one of those probes will make its way around the loop and back to the EtherFuse again. Duplicate packets in the network can lead to congestion, increasing the chance of probes getting dropped. Hence, EtherFuse drops all duplicate packets it detects. Figure 4 presents a flow chart of how loops are detected.

3.2.1 Building the Duplicate Detector

The EtherFuse's duplicate detector maintains the history of received packets in a hash table. However, it is desirable for the duplicate detector's hash table not to use chaining in order to simplify the implementation of the EtherFuse in hardware. In the following discussion, we assume that the range of the hash function that is applied to received packets is much larger than the size of the hash table. For example, the Ethernet packet's 32-bit CRC might be used as the hash code representing the packet. However, a hash table with 2^{32} entries would be impractical due to its cost. In such cases, a simple mapping function, such as mod the table size, is applied to the hash code to produce an index in the table. Since packets are represented by a hash code, the duplicate detector can report false positives. However, it is acceptable to have collisions with low probability since the EtherFuse will send a probe to verify that a forwarding loop exists.

There are two design alternatives to construct a duplicate detector with less entries than the hash function's range. The first alternative is to only store the timestamp in the hash table entry. In this case, two packets having different hash values may be mistaken as duplicates. This is because their two distinct hashes may map into the same location in the table. For this design alternative, false positives occur when detecting duplicate packets if two different packets with identical or different hashes map into the same location of the table. Assuming a uniform hash function, an upper bound for the probability of false positives occurring for a particular packet is given by Equation 1, where N is the number of entries in the duplicate detector, T is the time the packet's entry is kept in the duplicate detector before it expires, B is the network bandwidth, and F is the Ethernet's minimum frame size. Equation 1 computes the complement of the probability that the packet's entry in the hash table does not experience any collisions during its valid lifetime.

$$Pr = 1 - \left(\frac{N-1}{N}\right)^{\left(\lfloor \frac{T \times B}{F} \rfloor\right)} \quad (1)$$

The second design alternative to constructing the hash table is to include the packet's hash value in every entry along with the timestamp. In this case, two packets can be mistaken as duplicates only if they share the same hash value. An upper bound for the probability of false positives detecting duplicates for a particular packet is given by Equation 2, where K is the number of bits in the packet's hash. Similar to Equation 1, Equation 2 computes the complement of the probability that the packet's entry in the hash table does not experience any collisions during its valid lifetime.

$$Pr = 1 - \left(1 - 2^{-K}\right)^{\left(\lfloor \frac{T \times B}{F} \rfloor\right)} \quad (2)$$

However using this approach, EtherFuse can miss some duplicates. For example, if there exists a forwarding loop and a packet P_1 arrives at the EtherFuse, its hash will be recorded. Then by the time P_1 spins around the loop and before it arrives again at the EtherFuse, another packet, P_2, arrives first at the EtherFuse. If P_1 and P_2 have different hashes that hash into the same location in the hash table, the duplicate detector entry storing P_1's hash is replaced by P_2's hash. Since the duplicate detector records the packet's hash it will detect that P_2 is different than P_1 and not a duplicate. Later, when P_1 arrives again at the EtherFuse, its hash will replace P_2's hash in the duplicate detector without detecting a duplicate. Consequently, the EtherFuse will not detect that there is a loop. However, the probability of such a false negative is very low. An upper bound to this probability is given by Equation 3, where L is the latency around the loop. Equation 3 computes the probability that (1) packet P_1's hash gets replaced by one or more other packets' hashes before P_1 arrives again at the EtherFuse after cycling around the loop, and (2) the last packet of those which replaced P_1's hash entry in the duplicate detector has a different hash than that of P_1.

$$Pr = \left(1 - \left(\frac{N-1}{N}\right)^{\left(\lfloor \frac{L \times B}{F} \rfloor\right)}\right) \times \left(1 - 2^{-K}\right) \quad (3)$$

Figure 5 plots the probabilities in Equations 1, 2 and 3 with conservative values of the equations' parameters. The parameters were set as follows: T = 100ms, F = 64B, B = 10Gb/s, K = 32 and L = 10ms, where T is set to an order of magnitude more than L as a safety margin to minimize the chance of missing duplicates. In summary, the trade-offs between the two design alternatives are the following: Not including the packets' hashes in the hash table prevents false negatives when detecting duplicates. Thus, a

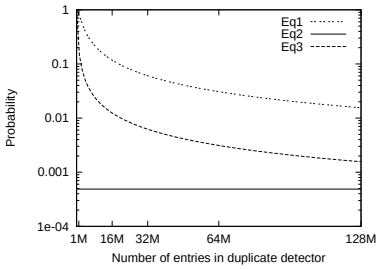


Figure 5: Plot of equations 1,2 and 3 illustrating the probability of false positives and negatives for the two design alternatives of the duplicate detector. Equation parameters were set as follows: T = 100ms, F = 64B, B = 10Gb/s, K = 32, and L = 10ms

forwarding loop is more likely to be detected as soon as the first duplicate is received. The down side of this alternative is that it suffers from a higher false positive rate when detecting duplicates. This leads to more non-duplicate packets getting dropped than in the second design alternative. However, for a duplicate detector with a sufficiently large number of entries, the false positives rate can be very low. For the second design alternative that includes the hash in every duplicate detector entry, it achieves a lower rate of false positives when detecting duplicates. However, this comes at a cost. First, false negatives occur when detecting duplicates. Thus, forwarding loop detection may get slightly delayed if a duplicate arrives at the EtherFuse but is not detected. Second, more memory is needed to store the hashes in the duplicate detector. Third, more per-packet computation is performed by the EtherFuse, specifically to compare the packet's hash to the corresponding hash in the hash table entry.

3.3 Mitigating Count to Infinity and Forwarding Loops

After detecting a count to infinity or a forwarding loop, the second phase is mitigating the problem and its effects.

For the count to infinity, if the EtherFuse detects BPDUs announcing increasing costs to a root R, it expedites the termination of the count to infinity by altering the message age field of any BPDUs announcing R to be the root. Specifically, it sets their message age field to *MaxAge*. However, this may not instantaneously terminate the count to infinity as Ethernet bridges may be caching other copies of the stale information. If there are other cached copies of the stale information, they will eventually reach the EtherFuse again, which in turn will increase their message age until eventually the count to infinity is terminated. Figure 3 shows how the EtherFuse handles a count to infinity. For handling the count to infinity, having more than one EtherFuse in a loop in the physical topology is not a problem as every EtherFuse can handle the count to infinity independently without side effects.

On the other hand, having more than one EtherFuse in the same loop in the event of a forwarding loop is problematic. Only one of those EtherFuses should cut the loop otherwise a network partition will occur. To handle this, EtherFuses collaborate to elect an EtherFuse that is responsible for breaking the loop. To do this, a probe carries the identities of the EtherFuses it encounters during its trip around the loop, that is, whenever an EtherFuse receives a probe originated by another EtherFuse, it adds its identifier to a list of EtherFuse identifiers in the probe.The EtherFuse's MAC address is used as its identifier. Also, the EtherFuse checks for its identifier in the list of identifiers in the probe. If it finds its own, then this probe has been through a loop. The EtherFuse drops such a probe as it is not the probe's originator. If the EtherFuse receives its own probe, it checks the list of EtherFuse identifiers attached to the probe. It drops the probe if its identifier is not the smallest in the probe's list of EtherFuse identifiers. On the other hand, if its identifier is the smallest in the list, the EtherFuse is elected to break the loop. It cuts the loop by blocking one of its ports that connects the loop. This way the network can continue operating normally even in the presence of a forwarding loop. However since physical loops exist in the network for redundancy and fault tolerance reasons, cutting them leaves the network vulnerable to partitioning due to future failures. So the EtherFuse tries to restore the network to its original topological state by unblocking its blocked port after a timeout period has passed. It does this hoping that the loop was a temporary loop formed due to ephemeral conditions. If the EtherFuse detects a loop again right after it tries to restore the network, then it knows that the loop still persists so it cuts the loop again. It retries this until it eventually gives up assuming this is a permanent loop. It then notifies the network administrator to take appropriate measures to fix the problem. Figure 4 shows how an EtherFuse handles a forwarding loop.

Since a forwarding loop may persist for a small duration until it is detected and corrected, forwarding table pollution may still occur. To speed recovery from forwarding table pollution, the EtherFuse sends BPDUs on both its ports with the topology change flag set. This will make bridges receiving this topology change information flush their forwarding tables and forward this topology change message to their neighbor bridges until it has spread throughout the network. This technique has its limitations though. This is because the IEEE 802.1D (2004) specification suggests an optimized technique for flushing entries from the bridge's forwarding table. This technique flushes entries for all the ports other than the one that receives the topology change message on the bridge. This technique is not mandatory but if it is implemented, there will be some cases in which the EtherFuse will not be able to eliminate the forwarding table pollution if the loop was not shutdown before pollution occurs. For example, the pollution shown in Figure 2(b) at port P1 cannot be fixed by an EtherFuse sitting along the loop B2-B3-B4. This is because B5 will receive the topology change messages at P1, the port with polluted forwarding table entries. Consequently, it will flush forwarding entries for port P2 and not P1. However, even if B5 does not flush the entries at P1, the polluted entries will be invalidated as soon as the end host H1 sends any packets or when those polluted forwarding table entries expire by reaching their timeout value.

4. ETHERFUSE IMPLEMENTATION

This section describes our prototype implementation of the EtherFuse. Then, it discusses the EtherFuse's memory and processing requirements, arguing that the EtherFuse can scale to large, high-speed Ethernets.

4.1 The EtherFuse Prototype

We implemented EtherFuse using the Click modular router [14]. Figure 6 shows how the different modules are put together to compose the EtherFuse in the Click modular router. The `FromDevice` module is responsible for receiving packets from a NIC into the EtherFuse. The `Classifier` module is responsible for classifying Ethernet packets based on their contents. In this configuration, it classifies them into either RSTP control packets which are sent to the `CTIChecker` module, or regular Ethernet data frames which are sent to the `LoopChecker` module. The `CTIChecker` module is responsible for handling count to infinity in the network, while the `LoopChecker` module is responsible for handling Ethernet forwarding loops. Ethernet frames are then pushed by the

`CTIChecker` and the `LoopChecker` modules to the other NIC of the EtherFuse using the `ToDevice` module. The suppressors, S1 and S2, allow the EtherFuse to block input and output ports, respectively. They are used by the EtherFuse to block both input and output traffic going through one NIC when a forwarding loop is detected.

In our implementation we used a packet's CRC as the hash of the packet's contents. We store the CRC in the duplicate detector hash table to reduce the number of dropped packets because of false positives. Timestamps in the duplicate detector have millisecond granularity. Thus, a timestamp is stored in 4 bytes.

4.2 Memory Requirements

The primary memory requirement for the EtherFuse is that of the duplicate detector which records the timestamps of frames it has received recently. The duplicate detector may also store the hashes of the packets. Every entry in the duplicate detector contains a hash with size C bytes, where C is equal to zero if the hash is not stored in the table, and a timestamp with size S bytes. The duplicate detector should have at least as many entries as the number of frames that make up the product of the maximum network bandwidth B and the latency L. Thus the minimum memory requirement M can be given by Equation 4, where F is the minimum Ethernet frame size.

$$M = \left(\left\lfloor \frac{L \times B}{F} \right\rfloor \right) \times (C + S) \quad (4)$$

The minimum frame size F is 64 bytes. Assuming the CRC is used as the packet's hash and a timestamp is 4 bytes, then C and S will be 4 bytes each. Using generous values of 100 milliseconds for L, and 10 Gbps for B would lead to M equal to 16MB. Thus the EtherFuse can easily scale to a large 10 Gbps Ethernet network.

4.3 Processing Overhead

The processing overhead of the EtherFuse is low. This is true even if the packet's hash is maintained in the duplicate detector along with the packet's timestamp, which would require more processing due to storing, loading and comparing hashes. Assuming the packet's hash is precomputed like if the CRC is used, then in the common case to handle a data packet one memory access is required to check whether the packet's hash exists in the duplicate detector, another memory access is required to write the hash into the duplicate detector, and another one to write the timestamp. This is assuming that at least 4 bytes can be read in a single memory access. In the unlikely event that the hash is matched, another memory access is needed to fetch the timestamp to check whether this hash is fresh or not. However, in conventional processors with data caches, fetching the hash from memory would lead to prefetching the timestamp as well if both are within the same cache line. In such cases, the access to the timestamp would have trivial cost. For BPDUs, they arrive at a much lower frequency than data packets, roughly on the order of 10 BPDUs per second even during a count to infinity. To handle a BPDU, the EtherFuse compares it against the 2 cached entries in the BCache. If a count to infinity is suspected, the BPDU is written into the BCache. Since a BPDU is 36 bytes, this requires at most 9 memory accesses for the comparisons and 9 memory accesses for the write. Since BPDUs arrive at a low rate, these operations can be easily handled in practice.

5. EXPERIMENTAL SETUP

This section describes the experimental settings used for the evaluation of the EtherFuse.

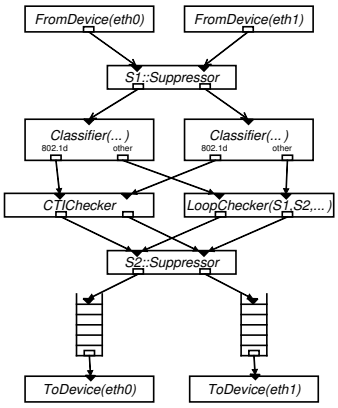
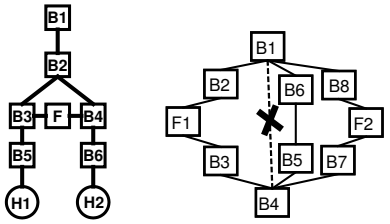


Figure 6: Block diagram of the EtherFuse implemented with the Click modular router



(a) Topology I. (b) Topology II.

Figure 7: Network topologies used used in the experiments.

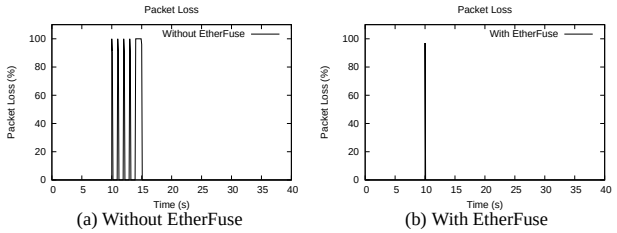
5.1 Hardware Platform

For our experiments we used the Emulab testbed [1, 22]. Specifically, we used machines with 3.0 GHz Xeon processors having 2GB of physical memory. Each machine had 6 Network Interface Cards (NICs). However, one of these NICs is used for the Emulab control network. In our experiments the machines were connected by 100 Mbps links to Cisco 6500 series switches.

The network topologies shown in Figure 7 are used for all of our experiments. In the figure, B's are Ethernet switches, F's are EtherFuses and H's are end hosts. For analysis, the EtherFuse collects statistics about the number of duplicate packets in the network. In the experiments where an EtherFuse is not used, a simplified device is substituted for the EtherFuse to collect the same statistics.

5.2 Software Components

All nodes in our experiments were running Fedora Core 4 with Linux kernel 2.6.12. Ethernet switches were implemented in software using Click. We used a configuration similar to the one prescribed in [14] for an Ethernet switch. However, when using RSTP switches, our configuration has two differences from that in [14]. First, we replaced the `EtherSpanTree` module which implements the legacy STP protocol with our own `EtherRSTP` module that implements the RSTP protocol. The RSTP module is responsible for maintaining the spanning tree of bridges, enabling or disabling switch ports based on their roles in the spanning tree. The second difference is that we updated the `EtherSwitch` module to support the functionality required for maintaining the switch's forwarding tables, including flushing and updating the tables in response to topology change events reported by the `EtherRSTP` module. We implemented the basic technique that flushes all forwarding tables in response to topology change events, not the optimized technique which does not flush the forwarding table for the port where the topology change event is received.



(a) Without EtherFuse (b) With EtherFuse

Figure 8: Timeline of packets loss for a 90 Mb/s UDP stream during count to infinity. Count to infinity starts at t=10 and no forwarding loop is formed.

6. EVALUATION

In this section we evaluate the effects of different Ethernet failures on software applications and show the effectiveness of the EtherFuse at mitigating these effects. For every class of failures, we study the fundamental effects using packet level measurements. Then, we use HTTP and FTP workloads to study the overall effects of the failures. The HTTP workload does not use persistent connections so the effects of the failures on TCP connection establishment can be studied. The FTP workload is used to study the effects of failures on TCP streams. We conduct multiple runs of each experiment because there is non-negligible variance between runs. This variance is due to non-deterministic interactions between the applications, the transport protocols, and the spanning tree protocols. However, in the cases where we characterize the network's behavior by a detailed timeline, it is only practical to present the results from one representative run. Results of the other runs are qualitatively similar.

The evaluation is organized as follows. Section 6.1 studies the effects of the count to infinity problem. Section 6.2 studies the effects of a single forwarding loop. Section 6.3 studies the effects of multiple, simultaneous forwarding loops. In both Sections 6.1, and 6.2 we use the topology shown in Figure 7(a) for our experiments, while in Section 6.3 we use the topology shown in Figure 7(b). Section 6.4 concludes with a discussion.

6.1 Effects of Count to Infinity

For the experiments in this section we modified the RSTP state machines such that its races do not lead to a forwarding loop in the event of a count to infinity. This is because we want to study the effects of the count to infinity in isolation without the forwarding loop.

6.1.1 Fundamental Effects

In this experiment, we characterize the packet loss in the network during the count to infinity. We use iperf to generate a 90 Mb/s UDP stream between the end hosts, which maintains high link utilization. Then, we measure packet loss at the receiving side of the UDP stream. iperf includes a datagram ID in every packet it sends and we instrumented it to maintain a history of the IDs it received to detect packet loss. Figure 8 presents a timeline of packet loss. The periodic heavy packet loss is caused by the oscillations of the network ports between blocking and forwarding during the count to infinity. It shows that during the count to infinity the network suffers from extended periods with near 100% loss rate. The EtherFuse substantially reduces these periods by terminating the count to infinity quickly.

6.1.2 Detection and Correction Time

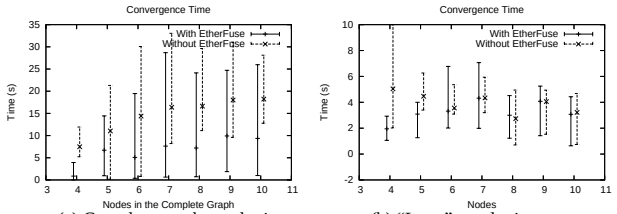
In this section, we study the time it takes the EtherFuse to detect and terminate a count to infinity using different network topologies. These experiments are based on simulations. This allows us to have global knowledge about the network and thus we can determine when the count to infinity has actually ended and the network has converged. We define convergence time as the time it takes all bridges in the network to agree on the same correct active topology. We use the BridgeSim simulator [17] but we have modified it to implement the latest RSTP state machines as specified in IEEE 802.1D (2004). In the simulator, bridges have desynchronized clocks so not all bridges start together at time zero. Instead each bridge starts with a random offset from time zero that is a fraction of the HelloTime. We have also added an EtherFuse implementation to the simulator. In our simulations, for each setting, we repeat the experiment 100 times and report the maximum, average, and minimum time values measured. We use a MaxAge of 20, a TxHoldCount of 3, and a HelloTime of 2 seconds.

In the experiment shown in Figure 9(a), we measure the convergence time in complete graph topologies after the death of the root bridge. For experiments with the EtherFuse, we used an EtherFuse for every redundant link. Notice that using EtherFuses for complete graph topologies cuts the average convergence time after a count to infinity by more than half.

Figure 10(a) shows the time it takes for the count to infinity to be detected by any EtherFuse in the network. We see that the EtherFuses detect the count to infinity very quickly. This is because for a count to infinity to be detected a bridge needs to transmit 2 consecutive BPDUs with increasing path cost that is higher than the cost it was announcing before the count to infinity. In this topology, all the bridges are directly connected to the root bridge and thus all bridges can detect the root's failure instantaneously. Hence, they immediately start using stale cached BPDU information, and start announcing different paths to the root which have higher cost. This constitutes the first increase in the path cost to the root. These stale BPDUs will trigger bridges to update their information and forward it to their neighbors. This constitutes the second increase in the path cost to the root, which is immediately detected by an EtherFuse. Thus, it takes two BPDU transmissions for some EtherFuses in the network to detect the beginning of the count to infinity. However it takes a much longer period of time for the network to converge. This is because bridges in the network have many redundant links and thus many alternate ports caching many copies of the stale information. Thus it takes time to replace all those stale copies of the information. Also ports in the Ethernet switches quickly reach their TxHoldCount limit due to multiple transmissions of the stale information. This further slows down the process of eliminating the stale information and makes the convergence time longer.

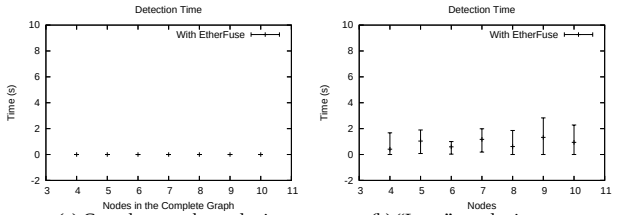
In the experiment shown in Figure 9(b), we measure the convergence time in "loop" topologies after the death of the root bridge. A loop topology is a ring topology with the root bridge dangling outside the ring. For these topologies we use a single EtherFuse. The EtherFuse connects the new root (the bridge that assumes root status after the death of the original root) to one of its neighbors. We note that for small loops, the EtherFuse is able to detect and stop the count to infinity quickly. However for larger loops, the EtherFuse becomes ineffective in dealing with the count to infinity. This is because the EtherFuse relies on observing two consecutive increases in the announced cost to the root. For loop topologies, this means the stale information must traverse around the loop twice. If the loop is large, the stale information will have reached its MaxAge before it gets detected by the EtherFuse.

Figure 10(b) also shows that the count to infinity is detected fairly quickly in the "loop" topologies. However, the termination of the count to infinity takes longer. This is because by the time the count to infinity has been detected, most of the bridges' ports have reached their TxHoldCount limit. Thus they are allowed to



(a) Complete graph topologies (b) "Loop" topologies

Figure 9: **Convergence time with and without the EtherFuse after the death of the root bridge.**



(a) Complete graph topologies (b) "Loop" topologies

Figure 10: **Detection time of count to infinity for EtherFuse.**

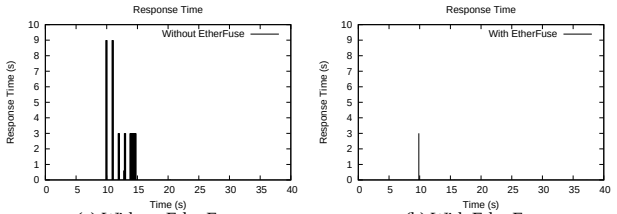
transmit only one BPDU per second and the convergence process is slowed down substantially.

6.1.3 Impact on HTTP

In this experiment we study the effects of count to infinity on web requests. We run the apache web server 2.2.0 on one of the end hosts and a program simulating web clients at the other end host. The client program generates HTTP requests to the web server with a constant rate of one request every 100 ms. The HTTP requests are HTTP GETs that ask for the index file which is 44 bytes. We kill the root bridge, B1, at time 10 to start the count to infinity. We repeat this experiment twice, once with the EtherFuse and another time without it and measure the response times of the web requests. Figures 11(a) and 11(b) show timelines of the measured response times of each web request before, during, and after the count to infinity with and without the EtherFuse. Note that before and after the count to infinity the response time is on the order of one millisecond. During the count to infinity, many requests have response times of 3 seconds and some even have response times of 9 seconds. This is due to TCP back-offs triggered by the packet loss during the count to infinity. TCP back-offs are especially bad during connection establishment, as TCP does not have an estimate for the round trip time (RTT) to set its retransmission timeout (RTO). Thus it uses a conservative default RTO value of three seconds. So if the initial SYN packet or the acknowledgment for this SYN gets dropped, TCP waits for three seconds until it retransmits. This explains the three second response times. If the retransmission is lost again TCP exponentially backs off its RTO to 6 seconds and so on. Thus we are able to observe requests having 9 second response times that are caused by 2 consecutive packet losses during connection establishment. In Figure 11(b), we note that the EtherFuse substantially reduces the period with long response times. This is because the EtherFuse is able to quickly detect and stop the count to infinity and thus reduce the period for which the network suffers from extensive packet loss. No connection in this experiment suffers consecutive packet losses during connection establishment.

6.1.4 Impact on FTP

In this experiment we study the effects of count to infinity on a FTP download of a 400MB file from a FTP server. The root bridge



(a) Without EtherFuse (b) With EtherFuse

Figure 11: **Timeline of response times of HTTP requests generated every tenth of a second under count to infinity. Count to infinity starts at t=10 and no forwarding loop is formed.**

	Transfer time
No failure	35.9s
Failure with EtherFuse	36.1s
Failure without EtherFuse	42.1s

Table 1: **Transfer times for the FTP transfer of a 400MB file.**

is killed during the file transmission and the total file transmission time is recorded. Table 1 shows the measured transmission times under count to infinity with and without the EtherFuse. We again note that transmission time in the presence of the EtherFuse is better as it ends the count to infinity early.

6.2 Effects of a Single Forwarding Loop

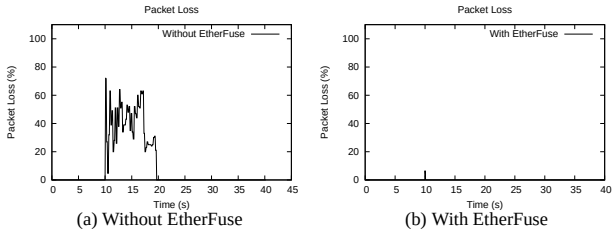
In this section, we study the effects of a single forwarding loop on applications and the performance of EtherFuse in mitigating those effects. We only focus on temporary forwarding loops because of two reasons. First, since the loops are temporary, they lead to transient interactions with the applications, which are often not obvious. Conversely, permanent loops render the network unusable leading to the unsurprising result of preventing applications from being able to make forward progress. Second, EtherFuse handles permanent loops the same way it handles temporary loops, so presenting the temporary loops case suffices.

We use count to infinity induced forwarding loops as an example of temporary forwarding loops. We modified the RSTP state machines such that its races always lead to a forwarding loop in the event of a count to infinity.

6.2.1 Fundamental Effects

Figure 12 shows a timeline of packet loss during the count to infinity induced forwarding loop. In this experiment a stream of UDP traffic flows from one host into another. Since the count to infinity reconfigures the network leading to the flushing of the bridges' forwarding tables and since the receiving end does not send any packets, bridges do not re-learn the location of the receiving end host. Thus, bridges fallback to flooding packets destined to the receiving end host. Thus, those packets end up trapped in the loop leading to network congestion and packet loss. This can be seen in Table 2. This massive packet loss leads to BPDU loss, extending the lifetime of the count to infinity. Consequently, this extends the duration of the forwarding loop leading to a longer period of network instability. When the EtherFuse is used the problem is corrected quickly.

To study the effects of having a temporary forwarding loop on a simple request/response workload we used ping between the two end hosts with a frequency of 10 pings per second. We ran this test for 50 seconds and introduced the count to infinity at time 10. Without the EtherFuse, we observed a 81% packet loss rate reported by ping. Note that there is no congestion in this test as the data rate



(a) Without EtherFuse (b) With EtherFuse

Figure 12: Timeline of packets loss using a 90 Mb/s UDP stream under a count to infinity induced temporary forwarding loop. Count to infinity starts at t=10.

EtherFuse	2
No EtherFuse	40815

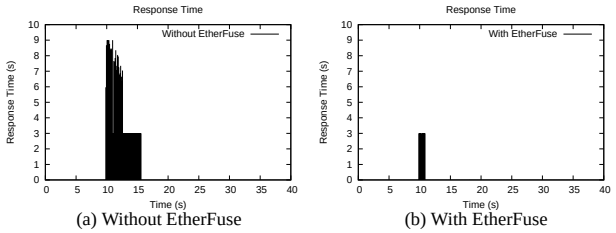
Table 2: Number of duplicate frames detected in the network for the UDP stream workload in event of having a forwarding loop.

is very low, and both end hosts are transmitting data so packets do not get trapped in the loop as in the experiment above. The main reason for the packet loss in this test is forwarding table pollution explained in Section 2.2.4. Specifically, in Figure 2 if a ping response from H1 causes the pollution, packets from H2 will not be able to reach H1 anymore. The pollution is fixed when the affected end host, H1, transmits a packet fixing the polluted forwarding table entry in B5. Thus, the pollution problem can last for a much longer period of time than that of the temporary forwarding loop. When the EtherFuse was used for the same experiment above, less than a 1% packet loss rate was reported by ping. This is because the EtherFuse quickly detects the forwarding loop, shutting it down and fixing any potential pollution by sending the topology change message that flushes the forwarding tables.

6.2.2 Impact on HTTP

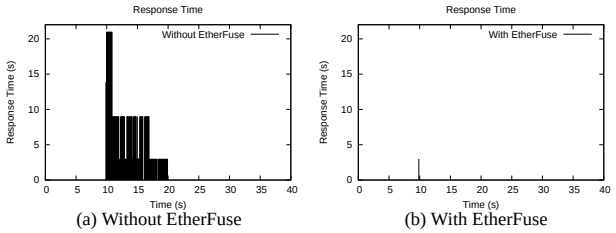
In this section, we repeat the experiments in Section 6.1.3, except that a forwarding loop is formed. Figure 13 shows a timeline of measured response times of web requests before, during and after the count to infinity induced forwarding loop. In the case of not having the EtherFuse, although the traffic in the network is minimal we note that having a forwarding loop hurts the response times of web requests. This is because although the connectivity is still available between the server and the client, packets coming from the client and the server into the forwarding loop pollute the bridges' forwarding tables. This leads to packet drops due to packet misforwarding and blackholing. Packet drops compounded with the TCP backoffs, especially during TCP connection establishment, lead to very high response times. In the case of EtherFuse, it detects and shuts down the forwarding loop very quickly so the disruption to the network operation is minimal. Note that for this workload pollution does not last for very long. This is because if a packet of an end host H causes pollution, the acknowledgment for this packet will not arrive, causing H to retransmit the packet fixing the pollution. If an acknowledgment packet from the server to the HTTP request causes the pollution, the server will send the response which will fix the pollution. Also, if the acknowledgment packet from the client to the HTTP response causes the pollution, either the next request or the connection tear down packet will fix the pollution.

Figure 14 shows the effects of having background broadcast traffic. We introduce a low rate broadcast stream of 100 Kb/s. Notice that in the no EtherFuse case, response times suffer substantially due to packet loss because of network saturation. This is because the broadcast packets get trapped in the loop leading to congestion. Also note that some web requests suffer from a 21 second response time. This is due to three consecutive packet drops in the connection phase leading to 3 exponential backoffs. When using the EtherFuse, it quickly detects both the count to infinity and the forwarding loop and cuts the loop to recover from this failure.



(a) Without EtherFuse (b) With EtherFuse

Figure 13: Timeline of response times of HTTP requests generated every tenth of a second under a count to infinity induced temporary forwarding loop. Count to infinity starts at t=10.



(a) Without EtherFuse (b) With EtherFuse

Figure 14: Timeline of response times of HTTP requests generated every tenth of a second under a count to infinity induced temporary forwarding loop. Count to infinity starts at t=10. A background broadcast traffic of 100 Kb/s is injected into the network.

To further understand the scenario, Table 3 shows the number of duplicate packets detected in the network. Note that a massive amount of duplicate packets are detected when there is background broadcast traffic and in the absence of the EtherFuse.

6.2.3 Impact on FTP

Table 4 shows the transfer times for a 400MB file over FTP when having a count to infinity induced forwarding loop. We note that when background broadcast traffic exists and in the absence of an EtherFuse many duplicate packets persist in the network as quantified in Table 5.

The main reason for the very long transfer time when the EtherFuse is not used is forwarding table pollution. Forwarding table pollution causes the FTP client to be cut off from the network for an extended period of time. In this case the pollution is very long lasting because it is caused by an acknowledgment by the client to a data packet sent by the server. The client then waits for the rest of the data packets to arrive for the rest of the file, but the server's packets cannot get through because of the forwarding table pollution. This causes TCP at the server to keep backing off. The problem only gets fixed later when the ARP cache entry for the FTP client expires at the FTP server forcing the server to send an ARP request for the client. Since ARP request packets are broadcast packets, they get flooded through the network and are not affected by forwarding tables. When the ARP request reaches the client, it makes the client send back an ARP reply which fixes the pollution and restores the connectivity to the client. When using the EtherFuse, this problem does not take place because after the EtherFuse detects and cuts the loop, it send the topology change message forcing bridges to flush their forwarding tables, including the polluted entries.

	Broadcast	No Broadcast
EtherFuse	9	1
No EtherFuse	57481	2

Table 3: Number of duplicate frames detected in the network for the HTTP workload in the event of having a forwarding loop.

	Broadcast	No Broadcast
No failure		35.9s
Failure with EtherFuse	37.2s	36s
Failure without EtherFuse	141s	140s

Table 4: Transfer times or a 400MB file over FTP.

6.3 Effects of Multiple Forwarding Loops

Multiple forwarding loops can occur due to the MaxAge induced forwarding loops as presented Section 2.2.2, or having two or more simultaneous failures of the failure types discussed in Section 2.2. In this section we choose the MaxAge induced forwarding loops as an example of multiple forwarding loops. To demonstrate the seriousness of having multiple forwarding loops, we construct an experiment using the network topology shown in Figure 7(b) that uses the STP protocol. We use a value of 2 for the MaxAge of the bridges in the network. This value is outside the prescribed range stated in the IEEE specification, but we use it so that we can generate the forwarding loops using only a few Emulab nodes. We connect an end host to $B3$ that sends a single broadcast packet. Then we measured the number of duplicate packets observed in the network every millisecond. We repeated this experiment twice, once with the EtherFuse, and another without. In the later case we see in Figure 15 that the packets exponentially proliferate until they saturate the network. This is because the CPUs of the Emulab nodes running network elements are saturated due to the processing of all the duplicate packets. When the EtherFuse is used we notice that the duplicates are eliminated from the network in 3 milliseconds. Roughly, one millisecond is spent on detecting duplicate packets, another millisecond for sending and receiving a probe, then another millisecond for the in transit duplicates to drain after the loop has been cut.

In summary, multiple forwarding loops can quickly render the network unusable due to exponential proliferation of duplicates. The EtherFuse is highly effective at detecting and correcting the problems.

6.4 Discussion

The EtherFuse is very effective at reducing the effects of a forwarding loop. Between the onset of a forwarding loop and its detection, the network may suffer from a very brief period of packet duplication. However, the EtherFuse is able to quickly stop packet duplication before it escalates into network congestion and packet loss. These benefits are achieved without changing the spanning tree protocols.

In contrast, while the EtherFuse is able to mitigate the effects of the count to infinity by reducing the spanning tree convergence time, the effects of the EtherFuse on count to infinity are not as immediate as for forwarding loops. The EtherFuse's ability to quickly end the count to infinity is constrained by the rate limit on BPDU transmission in the spanning tree protocols. Solutions that change the spanning tree protocols can eliminate the count to infinity and achieve much faster convergence. For example, in all of the scenarios discussed in Section 6.1.2, RSTP with Epochs [11] is able to converge in one round-trip time across the network.

	Loop/Broadcast	Loop/No Broadcast
EtherFuse	3	1
No EtherFuse	65578	19

Table 5: Number of duplicate frames detected in the network for the FTP workload in event of having a forwarding loop.

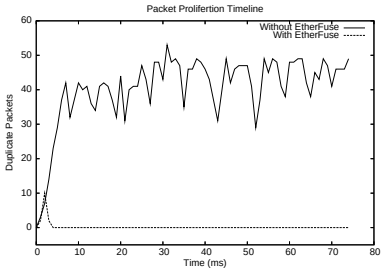


Figure 15: Timeline of number of duplicate packets observed by a network monitor after the formation of two forwarding loops and injecting an ARP request into the network.

7. RELATED WORK

The focus of this work is on mitigating the effects of Ethernet failures without changing the existing Ethernet infrastructure, including software, hardware, and protocols. In contrast, most previous work has focused on changing Ethernet's protocols to improve its scalability and performance. However, some hardware vendors employ techniques that try to enhance Ethernet's reliability.

Cisco employs two techniques to guard against forwarding loops, Loop Guard and the Unidirectional Link Detection (UDLD) protocol. None of these techniques is a part of the standard spanning tree protocols. Thus, not all vendors have these techniques implemented in their switches. Even Cisco does not have them implemented in all of their switches [6, 9]. Also, all those techniques require manual configuration which is error prone. For example both techniques are disabled by default on Cisco switches, so they need to be enabled first. Hence, having a single switch in the network that does not have or does not enable those features can leave the whole network vulnerable. This is because a single blocked port, erroneously transitioning to the forwarding state can make a forwarding loop that can render the whole network unavailable. Moreover, each of those techniques is limited in scope to a specific problem, so having one technique does not eliminate the need for the other. Finally, some kinds of forwarding loops can not be handled by any of those techniques, like the MaxAge induced forwarding loops and count to infinity induced forwarding loops.

The Loop Guard technique protects a network from BPDU loss induced forwarding loops. It prevents a blocked port from erroneously transitioning to the forwarding state when the port stops receiving BPDUs. Other than the shortcomings of this technique listed above, Loop Guard only works on point-to-point links. Thus, networks with shared links can be vulnerable to having forwarding loops even if the Loop Guard is used.

To guard against broadcast storms, broadcast filters are used in some Ethernet switches to suppress broadcast traffic to a certain level [4]. However, broadcast suppression suppresses broadcast packets indiscriminately once it reaches its maximum allowable level of broadcast traffic during a particular interval. Hence, duplicate broadcast packets may be allowed to get through before this cap is reached, saturating the filter, and then after the cap is reached legitimate broadcast traffic may get dropped.

UDLD is used to detect failures in which bidirectional links become unidirectional. The UDLD protocol disables the link to ap-

pear as if it is disconnected as the spanning tree protocol does not handle unidirectional links. UDLD relies on ports on both ends of a link exchanging keep-alive messages periodically. Missing keep-alive messages from one direction signal a failure in that direction. The inter-keep-alive message interval is manually configured by the network administrator. Again, other than the general shortcomings listed above this technique has a set of its own shortcomings. First, it needs ports on both ends of a link to support the UDLD protocol. Second, the keep-alive messages can get dropped in case of network congestion which can mislead the protocol to think that the link has failed.

Myers *et al.* [18] argued that the scalability of Ethernet is severely limited because of its broadcast service model. In order to scale Ethernet to a much larger size, they proposed the elimination of the broadcast service from Ethernet and its replacement with a new control plane that does not perform packet forwarding based on a spanning tree and provides a separate directory service for service discovery. Perlman [19] also argued that Ethernet has poor scalability and performance and proposed Rbridges to replace the current Ethernet protocols. Routing in Rbridges is based on a link state protocol to achieve efficient routing. Rbridges also encapsulate layer 2 traffic in an additional header that includes a TTL field to guard against problems from forwarding loops.

Several other previous works have addressed the inefficiency of spanning tree routing in Ethernet. SmartBridges [20] offers optimal routing using source specific spanning trees. LSOM [13] proposes using link state routing for Ethernet as well. Viking [21] delivers data over multiple spanning trees to improve network reliability and throughput.

RSTP with Epochs [11] modifies RSTP to eliminate the count to infinity problem and consequently eliminates count to infinity induced forwarding loops. That work studies the cause of count to infinity and the convergence time of RSTP and RSTP with Epochs in simulations. However, it does not consider the impact of the count to infinity problem on end-to-end application performance, nor does it consider other protocol vulnerabilities presented in this paper.

8. CONCLUSIONS

Although Ethernet is a pervasive technology, we have shown that it can suffer from serious problems due to simple local failures. These problems include extended periods of network-wide heavy packet loss, and in some cases complete network meltdowns. To address these problems, we introduced the EtherFuse, a new device that is backward compatible and requires no change to the existing hardware, software, or protocols. We implemented a prototype of the EtherFuse and used this prototype to demonstrate the effectiveness of the EtherFuse.

We have shown that the EtherFuse is very effective at reducing the effects of a forwarding loop. Between the onset of a forwarding loop and its detection, the network may suffer from a very brief period of packet duplication. However, the EtherFuse is able to quickly stop packet duplication before it escalates into network congestion and packet loss. The EtherFuse is also able to mitigate the effects of the count to infinity by reducing the spanning tree convergence time. However, the impact of the EtherFuse on count to infinity is limited by the design of the spanning tree protocols. Nevertheless, EtherFuse is able to provide its benefits in a way that is fully backward compatible.

9. REFERENCES

[1] Emulab - network emulation testbed. At http://www.emulab.net.

[2] A. Barnard. Got paper? Beth Israel Deaconess copes with a massive computer crash. Boston Globe, November 26, 2002.

[3] Beth Israel Deaconess Medical Center. Network Outage Information. At `http://home.caregroup.org/templatesnew/departments/BID/network_outage/`.

[4] Cisco Systems, Inc. Configuring Broadcast Suppression. At `http://www.cisco.com/univercd/cc/td/doc/product/lan/cat6000/sw_8_5/confg_gd/bcastsup.htm`.

[5] Cisco Systems, Inc. Internet Protocol Multicast. At `http://www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/ipmulti.htm`.

[6] Cisco Systems, Inc. Spanning-Tree Protocol Enhancements using Loop Guard and BPDU Skew Detection Features. At www.cisco.com/warp/public/473/84.html.

[7] Cisco Systems, Inc. Spanning Tree Protocol Problems and Related Design Considerations. At http://www.cisco.com/warp/public/473/16.html.

[8] Cisco Systems, Inc. Troubleshooting Transparent Bridging Environments. At www.cisco.com/warp/public/112/chapter20.pdf.

[9] Cisco Systems, Inc. Understanding and Configuring the Unidirectional Link Detection Protocol Feature. At www.cisco.com/warp/public/473/77.html.

[10] Cisco Systems, Inc. Understanding Rapid Spanning Tree Protocol (802.1w). At http://www.cisco.com/warp/public/473/146.html.

[11] K. Elmeleegy, A. L. Cox, and T. S. E. Ng. On Count-to-Infinity Induced Forwarding Loops in Ethernet Networks. In *IEEE Infocom 2006*, Apr. 2006.

[12] K. Elmeleegy, A. L. Cox, and T. S. E. Ng. Supplemental Note on Count-to-Infinity Induced Forwarding Loops in Ethernet Networks. *Technical Report TR06-878, Department of Computer Science, Rice University*, 2006.

[13] R. Garcia, J. Duato, and F. Silla. LSOM: A link state protocol over mac addresses for metropolitan backbones using optical ethernet switches. In *Second IEEE International Symposium on Network Computing and Apllications (NCA '03)*, Apr. 2003.

[14] E. Kohler, R. Morris, B. Chen, J. Jannotti, , and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[15] LAN/MAN Standards Committee of the IEEE Computer Society. IEEE Standard for Local and metropolitan area networks: Virtual Bridged Local Area Networks, 2003.

[16] LAN/MAN Standards Committee of the IEEE Computer Society. IEEE Standard for Local and metropolitan area networks: Media Access Control (MAC) Bridges - 802.1D, 2004.

[17] A. Myers and T. S. E. Ng. Bridgesim - bridge simulator. Version 0.03 is available from the author's web site, http://www.cs.cmu.edu/~acm/bridgesim/, May 2005.

[18] A. Myers, T. S. E. Ng, and H. Zhang. Rethinking the Service Model: Scaling Ethernet to a Million Nodes. In *Third Workshop on Hot Topics in networks (HotNets-III)*, Mar. 2004.

[19] R. Perlman. Rbridges: Transparent routing. In *IEEE Infocom 2004*, Mar. 2004.

[20] T. L. Rodeheffer, C. A. Thekkath, and D. C. Anderson. SmartBridge:A scalable bridge architecture. In *ACM SIGCOMM 2000*, Aug. 2000.

[21] S. Sharma, K. Gopalan, S. Nanda, and T. Chiueh. Viking: A multi-spanning-tree Ethernet architecture for metropolitan area and cluster networks. In *IEEE Infocom 2004*, Mar. 2004.

[22] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI'02)*, Dec. 2002.