Understanding and Mitigating the Effects of Count to Infinity in Ethernet Networks

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

***Abstract*— Ethernet's high performance, low cost, and ubiquity have made it the dominant networking technology for many application domains. Unfortunately, its distributed forwarding topology computation protocol – the Rapid Spanning Tree Protocol (RSTP) – is known to suffer from a classic count-to-infinity problem. However, the cause and implications of this problem are neither documented nor understood. This paper has three main contributions. First, we identify the exact conditions under which the count-to-infinity problem manifests itself, and we characterize its effect on forwarding topology convergence. Second, we have discovered that a forwarding loop can form during count to infinity, and we provide a detailed explanation. Third, we propose a simple and effective solution called RSTP with Epochs. This solution guarantees that the forwarding topology converges in at most one round-trip time across the network and eliminates the possibility of a count-to-infinity induced forwarding loop.**

***Index Terms*— Ethernet, Reliability, and Spanning Tree Protocols.**

I. Introduction

Ethernet[1] is the dominant networking technology in a wide range of environments, including home and office networks, data center networks, and campus networks. By far the most important reasons for Ethernet's dominance are its high performance-to-cost ratio and its ubiquity. Virtually all computer systems today have an Ethernet interface built in. Ethernet is also easy to deploy, requiring little or no manual configuration.

Even though Ethernet has all of these compelling benefits, mission-critical applications also require high network dependability. The dependability of Ethernet in the face of partial network failure is the focus of this study.

In existing Ethernet standards, packet flooding is used to deliver a packet to a new destination address whose topological location in the network is unknown. An Ethernet switch can observe 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 for a configurable period of time.

This research was sponsored by the NSF under CAREER Award CNS-0448546, by the Texas Advanced Research Program under grant No.003604-0078-2003, and by Cisco Systems, Inc. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, the state of Texas, Cisco Systems, Inc., or the U.S. government.

[1]In this paper, Ethernet always refers to the modern point-to-point switched network technology as opposed to the legacy, shared-media network technology.

To support the flooding of packets for new destinations and address learning, existing Ethernet standards rely on the dynamic computation of a cycle-free active forwarding topology using a spanning tree protocol. This active forwarding topology is a logical overlay on the underlying physical topology. Cycles in the underlying physical topology provide redundancy in the event of a link or switch failure. However, it is essential that the active forwarding topology be cycle free. Because Ethernet packets do not include a time-to-live field, they may persist indefinitely in a network cycle, causing congestion. Moreover, unicast packets may be misforwarded 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. Finally, when a link or switch failure disrupts the active forwarding topology, the network suffers from a period of packet loss. This packet loss lasts until the active forwarding topology is recomputed to bypass the failed component.

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 partial network failure. Today, the Rapid Spanning Tree Protocol (*RSTP*) [9] is the dominant spanning tree protocol. Unfortunately, RSTP may exhibit the count-to-infinity problem. During a count to infinity, the spanning tree topology is continuously being reconfigured and ports in the network can oscillate between forwarding and blocking data packets. Thus, many data packets may be dropped. Moreover, we have discovered that a temporary forwarding loop can form that may last until the count to infinity ends.

This paper presents an in-depth examination of this count-to-infinity problem in RSTP and provides a simple yet effective solution to it. The contributions of this study include:

- Characterize the exact conditions under which the count-to-infinity problem occurs in RSTP. *(Section III)*
- Identify the specific aspects of the RSTP specification that allow the count-to-infinity problem to occur. *(Section III)*
- Uncover harmful race conditions between the state machines defined in the RSTP specification. These race conditions in combination with a count to infinity can lead to a temporary forwarding loop. *(Section IV)*
- Provide a study of the count-to-infinity problem in RSTP under simple network topologies and different protocol parameter settings, demonstrating that protocol parameter tuning cannot adequately improve RSTP's convergence

time. *(Section V)*
- Propose and evaluate a simple yet effective solution that eliminates the count-to-infinity problem and dramatically improves the convergence time of the spanning tree computation upon failure to roughly one round-trip time across the network. *(Sections VI and VII)*

The rest of this paper is organized as follows. Section II provides an introduction to RSTP. Section III describes how count to infinity occurs in RSTP. Section IV explains how a count to infinity can lead to a forwarding loop in Ethernet. Section V studies the duration of a count to infinity. Section VI describes our solution to RSTP's count-to-infinity problem, the RSTP with Epochs protocol. Section VII evaluates this protocol. Section VIII discusses related work. Section IX concludes this paper.

II. The Rapid Spanning Tree Protocol (RSTP)

The Rapid Spanning Tree Protocol (RSTP) was introduced in the IEEE 802.1w standard and later revised in the IEEE 802.1D (2004) standard. It is the dominant Ethernet spanning tree protocol and the successor to the Spanning Tree Protocol (*STP*). It was derived from STP but designed to overcome STP's long convergence time that could reach up to 50 seconds [1]. In STP, each bridge maintains a single spanning tree path. There are no backup paths. In contrast, in RSTP, each bridge computes 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. The Spanning Tree

RSTP employs a distributed Spanning Tree Algorithm (*STA*) that computes a unique spanning tree over the network of bridges and connecting links. Under this algorithm, each bridge must have a unique ID. The spanning tree is rooted at the bridge with the lowest ID. The path through the spanning tree from any bridge to the root bridge is of minimum cost. To enable the network operator to select the root bridge, a bridge's default ID can be changed.

A bridge port, which connects a link to a bridge, has two main attributes, a *role* and a *state*. The port's role describes the port's place in the constructed spanning tree. Only three roles are relevant to this paper. First, a *root* port connects a bridge to its parent in the spanning tree. Second, a *designated* port connects a bridge to one or more children in the spanning tree. Thus, a bridge's parent is sometimes called its designated bridge. However, for clarity of presentation, we will call a bridge's parent in the spanning tree its parent bridge. Third, an *alternate* port is not a part of the spanning tree. It connects a bridge to a redundant link that provides a backup path to the root.

A port's state is either *forwarding* or *blocking*, depending on whether the port forwards data packets or blocks their flow. Generally, ports that are a part of the spanning tree are forwarding. However, during changes to the spanning tree, such ports may become blocking. An alternate port is always blocking.

B. Bridge Protocol Data Units

Bridges exchange topology information using messages called Bridge Protocol Data Units (*BPDU*s). Each bridge constructs its BPDUs based on the latest topology information that it has received from its parent bridge. Bridges send BPDUs to announce new information. In the absence of any new information, bridges still send a BPDU every *HelloTime* as a heartbeat. Heartbeat BPDUs are called *hello messages*.

A BPDU includes the ID of the root bridge and the cost of the bridge's path to this root. It also contains *MessageAge* and *MaxAge* fields. The MessageAge field is initialized to zero by the root bridge. When a non-root bridge sends a BPDU, it sets the MessageAge field to one more than the MessageAge of the BPDU that it last received from its parent. When the MessageAge exceeds the MaxAge, the message is dropped.

Each bridge uses a token-bucket algorithm to limit the rate of BPDU transmission per port. The bucket size is given by the bridge's Transmit Hold Count, abbreviated TxHoldCount. Specifically, each port has a counter, TxCount, that keeps track of the number of transmitted BPDUs. If TxCount reaches TxHoldCount, no more BPDUs are transmitted by the port during the current second. The token-bucket algorithm uses a token rate of one. In other words, the TxCount is decremented by one every second, unless its value is already zero.

A bridge needs to compare the BPDUs that it receives, based on the information that the BPDUs carry, so that it accepts and uses the best of these BPDUs. According to the IEEE 802.1D (2004) standard, BPDU M1 is *better* than BPDU M2 if:

1) M1 is announcing a root with a lower bridge ID than that of M2, or
2) Both BPDUs are announcing the same root but M1 is announcing a lower cost to reach the root, or
3) Both BPDUs are announcing the same root and cost but M1 was last transmitted through a bridge with a lower ID than the bridge that last transmitted M2, or
4) Both BPDUs are announcing the same root and cost, both BPDUs were last transmitted through the same bridge, but M1 was transmitted from a port with a lower ID than the port that last transmitted M2, or
5) Both BPDUs are announcing the same root and cost, both BPDUs were last transmitted through the same bridge and port on that bridge, but M1 was received on a port with a lower ID than the port that last received M2.

C. Building and Maintaining the Spanning Tree

The STA uses the information in the BPDUs to elect the root bridge and set the port roles on each bridge. Each port records the best information it received. The port that has received the best information, among all information received by all bridge ports, for a path to the root becomes the root port. Ports that receive worse information than they are sending become

designated ports. A port becomes an alternate port if it is not the root port and receives better information than it is sending.

If a root or alternate port has not received a BPDU in three times the *HelloTime*, the STA concludes that the path to the root through this port has failed and discards the information associated with this port. Physical link failures are detected even faster. If a bridge detects failure at its root port, it falls back immediately to an alternate port if it has any.

To avoid creating temporary forwarding loops, the blocking and unblocking of bridge ports during spanning tree reconfiguration must follow a particular order. Specifically, in the common case, before a designated port connecting a parent to a child can become forwarding, the child's designated ports must first be blocked. To impose this order, wherever a point-to-point link connecting a parent to a single child exists, RSTP relies on a hand-shake operation, called *sync*. When a blocked designated port wants to become forwarding, it requests permission from its child first. This is done by sending its child a BPDU with the *proposal* flag set. In response, its child typically blocks all its designated ports then responds to its parent with a BPDU with the *agreement* flag set. This operation cascades down the spanning tree because the blocked designated ports want to become forwarding again. For example, in Figure 2(a), if the link connecting bridges 2 and 3 did not initially exist, all the ports in the network would be forwarding as root or designated ports. If a link is added between bridges 2 and 3 and the sync operation is not performed, both ports at the two ends of the link can become forwarding simultaneously and a temporary forwarding loop is formed. The sync operation allows only one port to become forwarding at a time. This sync operation cascades downwards until it reaches a port that should be permanently blocked, which in the figure would be the port at bridge 4 connecting it to bridge 3.

D. Handling Topology Changes

A topology change can result in the reconfiguration of the spanning tree. Consequently, the port at each bridge that is used to forward to any given MAC address may have changed, thus requiring the invalidation of prior forwarding table entries. The STA implements this by making a bridge send a Topology Change (*TC*) message whenever it detects a topology change event. A topology change event arises when a blocked port becomes a forwarding port. The bridge sends such messages on all of its ports participating in the active topology. A bridge receiving a TC message forwards this message on all of its ports participating in the active topology except for the one that it received the TC message on. Whenever a bridge receives a TC message on one of its ports, it flushes the forwarding table entries at all of its other ports.

III. Count to Infinity in RSTP

A count to infinity can occur in RSTP when there is a cycle in the physical topology and this cycle loses connectivity to the root bridge due to a network failure. Figure 1 gives a simple example of a vulnerable topology. The path between bridge

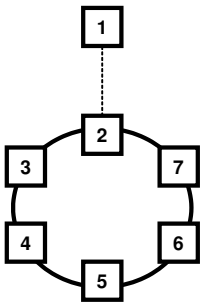
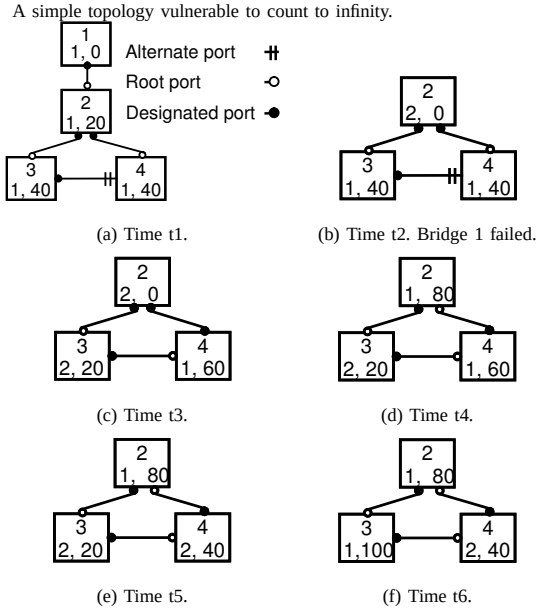


Fig. 1. A simple topology vulnerable to count to infinity.



(a) Time t1. (b) Time t2. Bridge 1 failed.

(c) Time t3. (d) Time t4.

(e) Time t5. (f) Time t6.

Fig. 2. An example of count to infinity.

1, the root, and bridge 2 does not have to be a direct link. A failure anywhere in this path can result in a count to infinity.

Specifically, the problem is bridges cache topology information from the past at their alternate ports, then use the information indiscriminately in the future if the root port loses connectivity to the root bridge. This topology information may be consistent or inconsistent with the current topology; we call it fresh or stale information respectively. A bridge using its cached information indiscriminately may end up using stale information. Then, the bridge may spread this stale information to other bridges via its BPDUs potentially resulting in a count to infinity.

To illustrate how a count to infinity occurs in RSTP, we first present an example. Then, we present a general proof.

A. An Example

First, we state four rules from the RSTP specification that are relevant to our example.

1) If a bridge can no longer reach the root bridge via its root port and does not have an alternate port, it declares itself to be the root. *(Clause 17.6)*
2) A bridge sends out a BPDU immediately after the topology information it is announcing has changed, e.g. when it believes the root has changed or its cost to the root has changed. *(Clause 17.8)*
3) A designated port becomes the root port if it receives a better BPDU than what the bridge has received before.

That is, this BPDU announces a better path to the root than via the current root port. *(Clauses 17.6 and 17.7)*

4) When a bridge loses connectivity to the root bridge via its root port and it has one or more alternate ports, it adopts the alternate port with the lowest cost path to the root as its new root port. *(Clauses 17.6 and 17.7)*

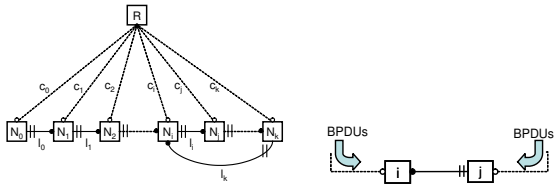
Now consider the example in Figure 2 showing a network of bridges. A box represents a bridge. The upper number in the box is the bridge ID. The lower two numbers represent the root bridge ID as perceived by the current bridge and the cost to this root. Link costs are all arbitrarily set to 20. Figure 2(a) shows the stable active topology at time t1. Figure 2(b) shows the network at time t2 when the link between bridge 1 and 2 dies. Bridge 2 declares itself to be the root since it has no alternate port (rule (1)). Bridge 2 announces to bridges 3 and 4 that it is the root (rule (2)). At time t3 bridge 3 makes bridge 2 its root as it does not have any alternate port. However, bridge 4 has an alternate port caching a path to bridge 1. Moreover, bridge 4 incorrectly uses this alternate port as its new root port. In other words, it makes bridge 3 its parent on the path to the now unavailable bridge 1 (rule (4)). This is because bridge 4 has no way of knowing that this cached topology information at the alternate port is stale. At time t4, bridge 4 announces to bridge 2 that it has a path to bridge 1, spreading the stale topology information and initiating a count to infinity (rule (2)). Bridge 2 makes bridge 4 its parent and updates the cost to bridge 1 to 80 (rule 3). At time t5 bridge 3 sends a BPDU to bridge 4 saying that bridge 2 is the root. Since bridge 3 is bridge 4's parent, bridge 4 accepts this information and sets its cost to bridge 2 to be 40. At time t6 bridge 2 sends a BPDU to bridge 3 saying that it has a path to bridge 1. Bridge 3 makes bridge 2 its parent, updating its cost to bridge 1 to be 100. The stale topology information about bridge 1 continues to go around the cycle in a count to infinity until either it reaches its MaxAge or it gets caught and replaced by the fresh topology information.

B. The General Case

We now give a general proof that whenever a network is partitioned, if the partition that does not contain the root bridge has a cycle, there exists a race condition that can result in the count-to-infinity behavior. The proof proceeds by first demonstrating that at least one bridge in the partition without the previous root bridge must declare itself the new root and start transmitting BPDUs. Its BPDUs will race with stale BPDUs, announcing the previous root, around the cycle. This race may lead to a count to infinity.

Claim 1: If a network is partitioned, the partition without the previous root bridge must contain a bridge that has no alternate port.

Proof: Consider the general network scenario illustrated in Figure 3(a). A dotted line represents a network path that may contain unknown intermediate hops. A solid line represents a direct bridge-to-bridge connection. Before the partition, R is the root bridge in the network. Every bridge N_x has a certain shortest path to R with a cost of c_x. Upon the partition, bridges N_0 to N_k form a partition that has no connectivity to R.



(a) A general network scenario. (b) A BPDU Race Condition.

Fig. 3. Illustrations for conditions necessary for a count to infinity.

The proof is by contradiction. Let us assume that bridges N_0 to N_k all have one or more alternate ports to R immediately after the partition. Consider bridge N_0. Since N_0 has at least one alternate port, it must be directly connected to another bridge in the partition, say N_1, which has an alternate path to R that does not include N_0. Without loss of generality, assume the BPDU sent by N_1 is better than the BPDU sent by N_0. Thus, N_0 has an alternate port through N_1. Similarly for N_1, it must have an alternate port to R via another bridge, say N_2, and N_2's BPDU is better than N_1's so N_1 has an alternate port through N_2. This argument applies till bridge N_{k-1}. However, since there is a finite number of bridges, N_k must obtain an alternate port to R via one of the bridges N_0 to N_{k-2}. However, this is impossible because N_k's BPDU is better than the BPDUs from all other bridges. Thus, we have a contradiction. ■

Because there exists at least one bridge in the partition that does not contain the previous root that has no alternate port, by RSTP (rule (1)), this bridge, when it detects that its root port is no longer valid, it must declare itself as the new root and begin sending BPDUs announcing itself as the root. These BPDUs will be flooded across the partition. The next claim shows that if the partition contains a cycle, then there exists a race condition such that if the BPDU arrives at a bridge with an alternate port via its root port first, stale topology information cached at its alternate port about the previous root will be spread into the network, creating a count to infinity. However if the fresh BPDU arrives via the bridge's alternate port, it will replace the stale information cached at the alternate port preventing the count to infinity from occurring.

Claim 2: If a network is partitioned, and the partition without the previous root bridge contains a cycle, a race condition exists that may lead to count to infinity.

Proof: From Claim 1, in the partition containing the cycle, one or more bridges without alternate ports must eventually declare themselves as root bridges and send their own BPDUs to the rest of the bridges in the partition. In addition before the partition, the cycle must contain one or more bridges with an alternate port to the root. This is because before the partition, assuming no forwarding loop exists, the cycle must be cut in the active forwarding topology by RSTP. An alternate port therefore exists at the link where the cycle is cut.

Now consider Figure 3(b) where the link between bridges i and j is where an alternate port exists in the cycle. Bridge i is connected to the rest of the loop with a root port on its left and has a designated port that links it to bridge j. Bridge j is connected to the loop by its root port on its right and connected to bridge i by an alternate port. After the partition, BPDUs from one or more bridges declaring themselves to be root will race around the cycle.

If such BPDUs are received by bridge j on its root port before its alternate port, bridge j will find that its alternate port has better cached topology information. Such information suggests a path to a superior root that is no longer reachable. Based on this stale information, bridge j will adopt its alternate port as its new root port. Then bridge j will start sending BPDUs conveying the stale information it has cached to bridges on its right. This is because bridge j believes that the topology information it has cached is better than the information it received from its neighbor bridge at its right. Afterwards, bridge j would get BPDUs on its new root port through bridge i from bridges declaring themselves to be root. Bridge j will then know that the topology information at its root port is stale and will accept the new topology information and also forward such new information to its right. This will result in a situation where fresh BPDUs chase stale BPDUs around the loop resulting in a count-to-infinity situation.

On the other hand if bridge j receives the fresh BPDUs from other bridges declaring themselves to be root on its alternate port first before receiving them on its root port, the stale topology information at the alternate port will be discarded and no count to infinity would occur. ■

Count to infinity may even occur without a network partition. For example if the loop in the physical topology loses its cheapest path to the root and picks another path with a higher cost. This new topology information will race around the loop until it reaches an alternate port caching stale, but better, information. Again this stale information will chase the new information around the loop in a count to infinity. This will keep going until the stale topology information reaches its MaxAge, or the cost reported by the stale information increases to exceed that of the new information. This is because the cost reported by the stale information increases while it is circling around the loop in a count to infinity.

IV. Count to Infinity Induced Forwarding Loops

The three key ingredients for the formation of a count to infinity induced forwarding loop are: First, count to infinity occurs around a physical network cycle. Second, during count to infinity, the fresh topology information stalls at a bridge because the bridge port has reached its TxHoldCount and subsequently the stale information is received at the bridge. As a result, the fresh information is eliminated from the network. BPDUs carrying stale information continue to propagate around the network cycle, and the count to infinity lasts until the stale information is aged out. Third, the sync operation that would have prevented a forwarding loop is not performed by a bridge because of a race condition and nondeterministic behavior in RSTP, allowing the forwarding loop to be formed.

In the following, we first precisely characterize the race condition and nondeterministic behavior in RSTP. Then, we provide a detailed RSTP event trace for an example network that serves as an existential proof of the formation of a forwarding loop during count to infinity in RSTP.

A. Race Condition

The RSTP specification contains a collection of state machines. RSTP state machines execute concurrently and communicate with each other via shared variables. The transitions between states are controlled by boolean expressions that often involve multiple variables. As stated in the specification, "The order of execution of state blocks in different state machines is undefined except as constrained by their transition conditions." Thus, many race conditions naturally occur between the RSTP state machines and some of them can be harmful.

In the following discussion, we refer to three of the RSTP state machines. First, the PORT INFORMATION state machine, a per port state machine, is responsible for handling the topology information arriving with new incoming BPDUs. Second, the PORT ROLE SELECTION state machine, a per bridge state machine, is responsible for checking if there are changes that need to be made to the port roles based on the new information received. Third, the PORT ROLE TRANSITION state machine, a per port state machine, is responsible for transitioning into the newly selected port role.

Claim 3: There exists a race condition between the PORT INFORMATION state machine and the PORT ROLE TRANSITION state machine when a bridge receives, from its parent at its root port, a BPDU that: (1) carries worse topology information than what the root port currently has and (2) does not cause the bridge to change its root port. This race allows the bridge to respond with a BPDU carrying the agreement flag without doing a sync operation. The race condition occurs in two cases:

(a) The received BPDU *has* the proposal flag set.
(b) The received BPDU *does not have* the proposal flag set.

The complete proof for Claim 3 is in the appendix. The proof for case (a) proceeds as follows: When a new BPDU arrives at a bridge's root port it gets handled by the PORT INFORMATION state machine. Then, the PORT ROLE SELECTION state machine gets executed. If the received BPDU has the proposal flag set and conveys worse topology information, like announcing a higher cost to the root, and that information does not change the port roles, a race condition occurs. Different executions are possible after the PORT ROLE SELECTION state machine completes execution, either the PORT ROLE TRANSITION state machine can execute at the root port or the PORT INFORMATION state machine can execute at designated ports. If the execution of the PORT INFORMATION state machine at the designated ports takes place first as intended, the ports' states are updated and the sync operation is performed. On the other hand, if the PORT ROLE TRANSITION state machine executes at the root port first, it will use a stale state of the other designated ports that allows the bridge to immediately respond with an agreement without doing the sync operation. This will allow a forwarding loop to form. The proof for case (b) is similar.

B. Formation of a Forwarding Loop: An Example

In this section, using a trace of protocol events, we show that the count to infinity in RSTP can lead to the formation of a forwarding loop. Table I shows a trace of protocol events after the failure of the root bridge, bridge 1, in the network shown in Figure 2. The first column of the table shows the time of occurrence for each event in increasing order. The second and

Time	BPDU Direction	BPDU Contents (Root, Cost[, Flags])	Comments
		Round 1	
t1	B2 → B3	2, 0	
t2	B2 → B4	2, 0	
		Round 2	
t3	B3 → B2	2, 20, Agreement	*Claim 3(b)*
t4	B3 → B4	2, 20	
t5			Block 4p2, B4 changes its root port, sync operation.
t6	B4 → B2	1, 60, Proposal	
t7			Unblock 4p3, new root port goes forwarding.
t8	B4 → B3	1, 60, Topology Change, Agreement	
t9	B4 → B2	1, 60, Topology Change, Proposal	
		Round 3	
t10	B4 → B2	2, 40, Topology Change, Proposal	
t11	B4 → B3	2, 40, Topology Change, Agreement	*Claim 3(b)*
t12			Block 2p3, proposal arrives from B4, sync operation at B2.
t13	B2 → B3	1, 80, Proposal	
t14	B2 → B4	1, 80, Agreement	
t15			Topology Change/Agreement arrives at B3 but with a better priority vector than the port's priority vector. Invalid agreement, ignored. *(Clauses 17.21.8 & 17.27)*
		Round 4	
t16	B3 → B2	1, 100, Topology Change, Agreement	
t17	B3 → B4	1, 100	
t18			B2 updates its state to be the root bridge, cannot propagate the information through its designated ports, 2p3 and 2p4, as they have reached their TxHoldCount.
t19			Agreement arrives at B4 but with a better priority vector than the port's priority vector. Invalid agreement, ignored. *(Clauses 17.21.8 & 17.27)*
		Round 5	
t20			Agreement arrives at B2 but with a better priority vector than the port's priority vector. Invalid agreement, ignored. *(Clauses 17.21.8 & 17.27)*
t21			BPDU from bridge 3 arrives at bridge 4, but no BPDU is sent to bridge 2 since 4p2 has reached its TxHoldCount.
		Round 6	
t22	B4 → B2	1, 120, Topology Change, Proposal	Occurs after a clock tick at B4 decrementing TxCount.
		Round 7	
t23			Reroot at B2, 2p4 is the new root port; sync, 2p3 is already blocked.
		Round 8	
t24	B2 → B3	1, 140, Topology Change, Proposal	Occurs after a clock tick at B2 decrementing TxCount.
t25	B2 → B4	1, 140, Topology Change, Agreement	Also occurs after a clock tick at B2 decrementing TxCount.
		Round 9	
t26			Unblock 4p2, agreement arrives.
t27	B3 → B2	1, 160, Topology Change, Agreement	*Claim 3(a)*
t28	B3 → B4	1, 160, Topology Change	
		Round 10	
t29			Unblock 2p3, agreement arrives.

TABLE I

AN EXAMPLE SEQUENCE OF EVENTS, AFTER FAILURE OF THE ROOT BRIDGE IN FIGURE 2, THAT LEADS TO A FORWARDING LOOP.

third columns are used if the event is a BPDU transmission. The second column shows the bridges sending and receiving the BPDU. The third column shows the contents of the BPDU. The fourth column shows additional comments describing the event. Rows in the table are grouped into rounds, where events in each round are triggered by either messages from the previous round or a clock tick. We use the notation ipj to name the port at bridge i connecting it to bridge j. We also use a fixed-width font to refer to state machine variables and a fixed-width font with all capital letters to refer to state names.

Assume that bridge 1 has died right after bridge 2 has sent out a hello message but before its clock has ticked. Thus, the TxCount is one for ports 2p3 and 2p4 and zero for ports 3p2, 3p4, 4p2, and 4p3. Also assume that bridges use a TxHoldCount value of 3. Thus, each port can transmit at most 3 BPDUs per clock tick.

Round 1: After the death of bridge 1, bridge 2 will declare itself to be the new root and propagate this information via BPDUs at t1 and t2.

Round 2: At t3, bridge 3 will send back an agreement to bridge 2 as the information received by bridge 3 is worse than the information it had before *(Claim 3(b))*. At t4, bridge 3 will pass the information it received from bridge 2 to bridge 4. Since bridge 4 has a cached path at its alternate port to the retired root, bridge 1, it will believe this stale information to be better than the fresh information it received at 4p2 from bridge 2. Thus, bridge 4 decides to use this stale information and make its alternate port its new root port. This change of the root port involves a sync operation that temporarily blocks 4p2 until a proposal/agreement handshake is done with bridge 2, as described in *Clauses 17.29.2 & 17.29.3* of the RSTP specification. The temporary blocking of 4p2 occurs at t5. Then bridge 4 sends a BPDU to bridge 2 at t6 informing it that bridge 4 has a path to a better root bridge, bridge 1, with cost 60 and proposes to be bridge 2's parent. After blocking 4p2, it is now safe for bridge 4 to unblock its new root port so it unblocks 4p3 at t7. Since a new port, 4p3, has become forwarding, this constitutes a topology change event and thus bridge 4 sends a topology change message to bridge 3 at t8. Bridge 4 also sends another topology change message to bridge 2 at t9.

Round 3: At t10, the information from bridge 3 announcing bridge 2 to be the root arrives at bridge 4. Bridge 4 then passes this information to bridge 2. Since 4p2's `proposing` flag is still set, the new message is sent along with a proposal flag. Now port 4p2 has reached its TxHoldCount limit. 4p2 has sent three messages at t6, t9 and t10. Thus this port can not send any more BPDUs during this clock tick. Then bridge 4 sends back an agreement to bridge 3 at t11 for the information it received since this information is worse than what it had

(Claim 3(b)). At t12 bridge 2 receives the proposal along with the new information from bridge 4 and makes 2p4 its new root port in response to the new information. This leads to bridge 2 performing a sync operation blocking 2p3. Then at t13, bridge 2 passes on the new information to bridge 3 proposing to be bridge 3's parent. At t14, bridge 2 responds to bridge 4's proposal with an agreement, notifying bridge 4 that it agrees to bridge 4 being its parent. Note that now both ports 2p3 and 2p4 have reached their TxHoldCount limit. 2p3 has sent a hello message before bridge 1 died, then two more messages at t1 and t13. 2p4 has sent a hello message as well and two more messages at t2 and t14. Thus, both ports cannot send any more BPDUs during this clock tick. At t15, bridge 3 receives the topology change/agreement sent by bridge 4 at t8. However, this received BPDU is sent through a root port with better information than that stored at 3p4. Thus the message is discarded based on *Clauses 17.21.8 & 17.27* of the RSTP specification.

Round 4: When bridge 3 receives the proposal sent by bridge 2 at t13, it replies with an agreement at t16. This is because the information bridge 3 received is better than what it had before, so the `agree` flag does not get reset by `betterorsameinfo()` *(Clauses 17.21.1)*. When 3p2 enters the `SUPERIOR_DESIGNATED` state in the PORT INFORMATION state machine when it receives the new information *(Clause 17.27)*. Note that the agreement sent at t11 sets the `synced` flag of 3p4 to true. Bridge 3 also passes on the information to bridge 4 at t17. Then at t18 bridge 2 receives the information sent at t10 which makes it believe that it is the root bridge. However, it can neither pass the information to bridge 3 nor send back a response to the proposal that came along with the new information from bridge 4. This is because both 2p4 and 2p3 have reached their TxHoldCount limit. As a result, the fresh information that conveys that bridge 2 should be the root is stalled at bridge 2. At t19, bridge 4 receives the agreement sent by bridge 2 at t14. However, this received BPDU is sent through a root port with better information than that stored at 3p4. Thus the message is discarded based on *Clauses 17.21.8 & 17.27* of the RSTP specification.

Round 5: Similarly at t20, bridge 2 receives a stale agreement sent at t16 and thus the stale agreement gets discarded. At t21, bridge 4 receives the BPDU sent at t17. But since 4p2 has reached its TxHoldCount limit, BPDU transmission to bridge 2 is not allowed.

Round 6: When bridge 4's clock ticks at t22, bridge 4 passes the information it received from bridge 3 to bridge 2. Bridge 4 also includes the proposal flag as it never received a valid agreement from bridge 2 and thus the `proposing` flag is still set at 4p2.

Round 7: At t23, the stale information from bridge 4 conveying that bridge 1 is the root arrives at bridge 2 and eliminates the only copy of the fresh information stalled at bridge 2 that conveys bridge 2 is the root. Subsequently, only the stale information conveying bridge 1 is the root remains in the network until it is aged out. This stale information causes bridge 2 to believe again that bridge 4 is its parent and that port 2p4 is its new root port, this causes bridge 2 to do a sync operation. Port 2p3 is already blocked, and the sync operation does not change that. Since 2p3 has reached its TxHoldCount limit, it cannot send the new information along with the proposal BPDU until the clock ticks.

Round 8: When bridge 2's clock ticks at t24, it sends the proposal along with the new information to bridge 3. Also after bridge 2's clock ticks, it sends the agreement to bridge 4 at t25 for the proposal sent at t22.

Round 9: Bridge 4 receives the agreement from bridge 2 at t26 causing it to unblock 4p2. At t27, bridge 3 sends the agreement to bridge 2 responding to the proposal sent at t25 by bridge 2. Although the received information is worse than the information bridge 3 had earlier, it sends the agreement right away without doing a sync operation *(Claim 3(a))*. Bridge 3 also passes the new information to bridge 4 at t28. This makes port 3p4 reach its TxHoldCount limit based upon messages sent at t4, t17 and t28.

Round 10: The agreement sent at t27 reaches bridge 2 at t29 causing bridge 2 to unblock 2p3. All ports in the network cycle are now forwarding. Thus a forwarding loop is created.

From this point on until the end of the count to infinity, the BPDUs will all convey that bridge 1 is the root. None of them will have the proposal flag set. No bridge will perform any sync operation. Thus the forwarding loop will persist until the count to infinity ends when the stale information conveying that bridge 1 is the root is aged out.

V. How Long Does Count to Infinity Last?

One may think that count to infinity can last for at most a few milliseconds as the stale BPDUs are dropped after they traverse at most MaxAge number of bridges around the loop. Unfortunately BPDUs may traverse bridges slowly. It is difficult to predict the duration of a count to infinity as many factors are involved. However, bounds for this duration can be given. Count to infinity can terminate very quickly, if the fresh topology information catches the stale information and eliminates the stale information from the network. Otherwise if the stale information is allowed to persist till it reaches its MaxAge, then a count to infinity would terminate after (MaxAge $\times$ time to cross a single bridge).

A. Maximum Duration of Count to Infinity

A count to infinity must end when the stale information is discarded due to reaching the MaxAge. Thus the stale information can cross at most MaxAge hops. Topology information in a BPDU can reside in memory at a bridge for at most (3 $\times$ HelloTime) unless it gets refreshed by a new incoming BPDU *(Clauses 17.17.6 & 17.21.23)*. Therefore the theoretical upper bound for stale information to stay in the network is (3 $\times$ HelloTime $\times$ MaxAge). Since a count to infinity can last as long as there is stale information in the network, then the maximum lifetime of a count to infinity is (3 $\times$ HelloTime $\times$ MaxAge). For example, by default HelloTime is 2 seconds and MaxAge is 20. With these values, a count to infinity could last for 120 seconds.

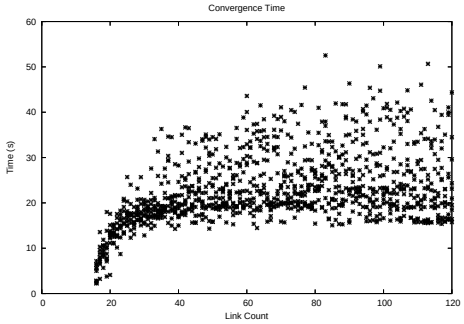


Fig. 4. Convergence time in a network of 16 bridges after failure of the root.

B. Slow Count to Infinity Termination

The duration of a count to infinity can approach the theoretical maximum because of three factors. First, BPDUs may stall at bridges because of the TxHoldCount limiting their rate of transmission. Second, BPDU packet loss can slow the propagation of topology information. Third, when bridges have multiple alternate ports, stale information can persist longer.

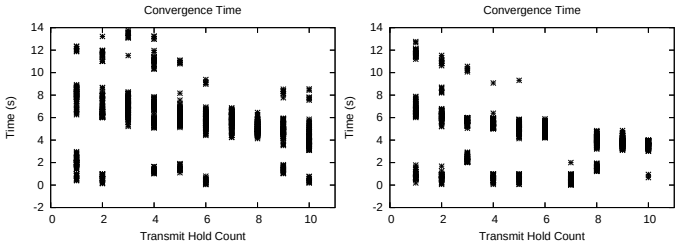
When a bridge loses connectivity through its root port to the retired root bridge, it fails-over to each of its alternate ports one at a time. Each fail-over to an alternate port lasts until the bridge realizes that the information cached at the new root port is stale. For a bridge to recognize that its root port is not valid, it needs to receive fresh information at its root port. Then, it will fail-over to an alternate port. The TxHoldCount slows the arrival of fresh information because it rate limits BPDU transmission. Thus it takes a long time for a bridge to try all its alternate ports.

While the bridge uses this stale cached information, it also transmits the stale information to its children. When stale information is received by a bridge, it is assumed to be fresh. Hence, it is allowed to be cached for ($3 \times$ HelloTime) seconds at the receiving bridge. This further perpetuates the lifetime of the stale information in the network.

C. Measured Duration of Count to Infinity

In the previous section we presented an upper bound for the count to infinity. In this section we measure the actual duration of the count to infinity under several simple network topologies using simulations. We simulate a network of 16 bridges that is initially configured in a ring topology. Then we randomly add redundant links to increase the topological complexity until we reach a fully connected graph. After adding each link we simulate the failure of the root bridge and measure the convergence time. What we mean by convergence time is the time, measured in seconds, after which all the bridges in the network have agreed on the same correct forwarding topology. The network converges after a count to infinity has ended. Since the simulator has global knowledge about the network topology, it can accurately measure the convergence time.

In our experiments throughout the paper we use a simulator we wrote [3] that is based on the simulator used by Myers *et al.* [11], but implements the IEEE 802.1D (2004) specification. The simulator uses a MaxAge value of 20, HelloTime of 2 seconds and a TxHoldCount of 3 unless otherwise stated. The simulator has desynchronized bridge clocks that is not all



(a) Complete graph topology, 4 bridges. (b) "Loop" topology, 4 bridges.

Fig. 5. Convergence time after failure of the root varying the TxHoldCount.

bridges start together at time zero. Instead each bridge starts with a random offset from time zero that is a fraction of the HelloTime. Bridges are connected to each other by links with 100 microseconds of total delay (propagation and transmission delay). Only protocol BPDU packets are simulated. No user data packet traffic is simulated.

Figure 4 presents the convergence times measured. For every number of links we repeat the experiment 10 times and report the measured convergence times under the count to infinity. We note that adding more redundant links dramatically increases the convergence time, to reach 50 seconds in one of the experiments. This is because adding more redundant links results in more alternate ports per bridge. Those alternate ports extend the duration of a count to infinity as explained in Section V-B and increase the convergence time.

D. Effects of Tuning RSTP Parameters on the Convergence Time

From Sections V-A and V-B, we see that RSTP's convergence time during a count to infinity is mainly influenced by three parameters: HelloTime, MaxAge, and TxHoldCount. HelloTime cannot be decreased below one second according to the IEEE 802.1D (2004) specification and decreasing it would result in bridges transmitting more BPDUs making ports reach their TxHoldCount more quickly. Decreasing the MaxAge can reduce the duration of the count to infinity but it would also reduce the maximum spanning tree height reducing Ethernet scalability. Finally, one may think that by increasing the TxHoldCount, the duration a stale BPDU can persist in a network should be proportionally reduced. Unfortunately, this is not the case in reality.

To illustrate why tuning the TxHoldCount does not reduce the the convergence time to satisfaction, we simulate a fully connected network of 4 bridges and measure the convergence time after the death of the root bridge. Figure 5(a) shows the convergence times for ten runs when varying the TxHoldCount according to the value range allowed by the RSTP standard. For every value of the TxHoldCount we repeat the experiment 100 times and report the measured times. We can see that the convergence time exhibits a multi-modal behavior. Even when the TxHoldCount is increased to 10, the worst case convergence time is still 8 seconds, not the 10 times improvement one might expect when comparing to a TxHoldCount of 1. Clearly, the benefit of increasing TxHoldCount is non-linear and limited. This is because once the TxCount reaches the TxHoldCount limit, it gets decremented by one every second allowing for only one BPDU to be transmitted per second irrespective of the TxHoldCount value.

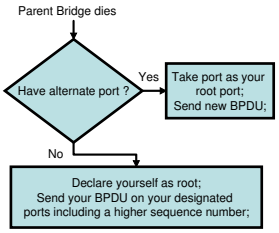


Fig. 6. Handling the death of the parent bridge.

Figure 5(b) shows the measured convergence times for a simpler topology, namely the topology in Figure 2(a). Even in this simple topology, increasing the TxHoldCount does not dramatically improve convergence time.

VI. RSTP with Epochs: Extending RSTP to Eliminate Count to Infinity

RSTP with Epochs is an extension to RSTP. It relies on introducing sequence numbers in the BPDUs. The root bridge adds a sequence number to each BPDU it generates. Then, the other bridges generate and transmit their own BPDUs based on the latest root's BPDU and including the root's latest sequence number. The purpose of these sequence numbers is to identify stale BPDUs or stale cached topology information from a retired root.

Sequence numbers by themselves are not sufficient. For example, consider in a network of bridges where there is the old root bridge A and a new bridge B with lower bridge ID than A that has just joined the network. Bridge B is now eligible to become the root, so when it receives a BPDU from A, it starts sending out its own using a sequence number higher than the one in A's BPDU. This is to override A's BPDUs and assert itself as the new root causing A to back-off. However, by the time B's BPDU reaches A, A may have sent out one or more BPDUs having higher sequence numbers. Hence, A will view B's BPDUs as stale. Consequently, A will not back off, and the network will not converge.

Using epochs solves this problem. An epoch is an interval starting when the true root bridge achieves root status and ending when another bridge contending for root status. Another bridge will contend for root status because it did not hear from the previous root or because it finds its bridge ID to be lower than that of the previous root. A bridge may not hear from the previous root if the previous root has retired, or the root may still be reachable but the contending bridge has lost its path to the root without having any other alternate ports. A bridge may find it has a lower bridge ID than the root because it has just joined the network and its bridge ID is lower than the current root's bridge ID, making it eligible to be the new root. If the previous root has retired and the contending bridge is eligible to be the root, the new root will use a sequence number higher than the highest sequence number it received from the retired root signaling a new epoch with a new root bridge. If the old root is reachable and is still eligible to be the root, it pumps up its sequence number to override the contending bridges' sequence numbers to re-take the network and this signals a new epoch as well but with the same root bridge as in the previous epoch. Each bridge has a local representation of an epoch with an interval of sequence numbers it heard from the same root bridge. The interval is represented by two sequence numbers, FirstSeqno and CurrentSeqno. FirstSeqno is the first sequence number this bridge has heard from the current root. CurrentSeqno is the current or latest sequence number the bridge has heard from the root. Back to the example given above, epochs allow the new root B to catch up with the old root's sequence numbers to eventually be able to take over the network. When B's BPDU reaches A, A may have already sent BPDUs with higher sequence numbers, but since B's BPDU sequence number lies within the interval representing the current epoch, A realizes that B coexists with it in the same epoch and thus it backs away. Section VI-A presents RSTP with Epochs. Then Section VI-C further discusses the operation of the protocol.

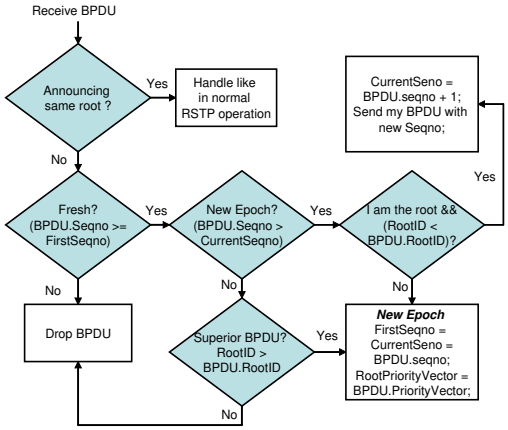


Fig. 7. Handling the reception of a BPDU in RSTP with Epochs.

A. Protocol Definition

The periodic BPDUs sent by the root have increasing sequence numbers (BPDU.Seqno), where the period is typically a HelloTime. The sequence number is incremented by the root bridge at the beginning of each period. Non-root bridges generate their BPDUs including the root's latest sequence number.

Each bridge records two values, FirstSeqno and CurrentSeqno, the first and last sequence numbers, respectively, that it has received from the current root bridge. These two sequence numbers define the current epoch. The purpose of this epoch is to identify stale BPDUs. A BPDU with a sequence number less than the recorded first sequence number must be a stale BPDU belonging to an earlier epoch.

As shown in Figure 6, when a bridge detects disconnection from its parent, it first checks to see if it has any alternate ports. If it does, it adopts one of these alternate ports as its new root port. However, if the bridge does not have any alternate ports, it declares itself as the new root and starts broadcasting its own BPDUs that have a sequence number larger than the last sequence number that it received from the old root.

Figure 7 explains the handling of the receipt of a BPDU for RSTP with Epochs. Bridges disregard the sequence numbers when comparing BPDUs declaring the same root. However, if a BPDU arrives declaring a different root than the one perceived by the bridge, the bridge checks if the BPDU's sequence number is larger than the last recorded sequence number for the perceived root. If this is the case, it signals the

beginning of a new epoch. The new epoch has a different root declared by the received BPDU. The first and last sequence numbers are set to the sequence number reported by the received BPDU. On the other hand, if the sequence number reported by the BPDU is larger than or equal to the first recorded sequence number but smaller than or equal to the largest recorded sequence number of the current root, the bridge with the lowest ID, among the ones declared by the BPDU and the current root, is deemed superior; and it is the one accepted by the bridge as the current root.

If a bridge receives a BPDU declaring another bridge with an inferior bridge ID to its own as the root, the bridge starts sending BPDUs declaring itself as the root. These BPDUs are given a sequence number that is larger than that received from the bridge with the inferior ID. When one of these BPDUs reaches the old root bridge with the inferior ID, it will stop declaring itself as the root.

Sequence numbers can wrap around. The way to deal with that is to consider zero as bigger than the largest sequence number. A side effect of doing that is when a new bridge joins the network starting off with sequence number zero, it may be able to temporarily take over the network, asserting itself as the new root, although it has a bridge ID higher than the legitimate root. When the legitimate root receives the new bridge's BPDU, it can then increase its sequence number and re-take the network. This may result in a brief period of disconnectivity. A solution to this problem is to make a new bridge joining the network listen for BPDUs for a random period. If it receives a BPDU from a superior root, it should not send its own BPDU. If no better BPDUs are received the new bridge can then start sending its own BPDU declaring itself to be the new root.

B. Interoperability with Legacy Bridges

The basic mechanism for RSTP with Epochs to interoperate with RSTP and STP is similar to that used by RSTP to interoperate with STP. First, RSTP with Epochs should be assigned a new protocol version number. A BPDU sent by a bridge carries the version number of the corresponding protocol used. A BPDU with an unknown version number will be discarded by the receiving bridge. At start up, a RSTP-with-Epochs bridge will try sending RSTP-with-Epochs BPDUs. If the network peer is a legacy bridge, these BPDUs will be ignored. Eventually, the RSTP-with-Epochs bridge will receive legacy BPDUs from the legacy peer bridge, at such time it can recognize the protocol used by the peer and fall back to the appropriate legacy protocol. To translate a RSTP-with-Epochs BPDU into a legacy BPDU, the epoch sequence number is simply stripped from the BPDU.

C. Discussion

In absence of a count to infinity, both RSTP and RSTP with Epochs generate the same topology change events and thus generate the same number of BPDUs signaling the topology change events. This is because a topology change event occurs when a port becomes forwarding and both protocols converge to the same topology, switching the same ports to forwarding. Thus, both protocols generate the same topology change events. In case of a count to infinity in RSTP, some ports may become forwarding temporarily generating some extra topology change events as in Figure 2.

The disadvantage of RSTP with Epochs when compared to RSTP is the small overhead that can result from its comparative pessimism. To elaborate, let us reconsider the topology in Figure 1. Suppose the link between bridge 2 and 3 dies. Under both protocols, bridge 3 will emit a new BPDU. The difference is, in RSTP, the propagation of this BPDU will be stopped once it reaches bridge 5 because bridge 5 has an alternate port to the root via bridge 6. In effect, by default RSTP assumes that the root bridge is still alive. In contrast, in RSTP with Epochs, this BPDU creates a new epoch and thus is better than the cached information at the alternate port at bridge 5. Consequently the propagation will not be stopped until it reaches bridge 1. In effect, RSTP with Epochs pessimistically assumes that the root bridge is inaccessible.

VII. Evaluating RSTP with Epochs

To evaluate RSTP and RSTP with Epochs we used the simulator described in section V-C. We extended it to include the RSTP-with-Epochs implementation.

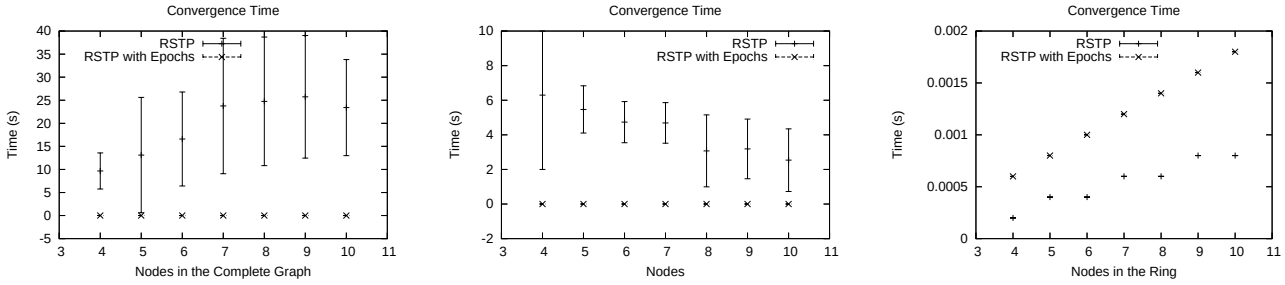
We first evaluate the convergence times of both protocols. Then, the packet overhead of both protocols is studied. Finally, we study how count to infinity can saturate a bridge's maximum BPDU transmission rate limit (i.e. the TxHoldCount), thus preventing the timely announcements of other BPDUs.

A. Comparing Convergence Times of Both Protocols in the Event of Failure

In this section we compare the convergence times of RSTP and RSTP with Epochs in the event of failure in three families of topologies. For each family of topologies we vary the number of bridges in the network and measure the corresponding convergence time. For each data point we repeat the experiment 100 times and report the range of values measured.

In the first experiment we simulate a set of complete graphs, varying the number of bridges in the network. In each run we kill the root bridge and measure the time it takes for the network to converge under both protocols. Figure 8(a) shows the convergence times measured. It presents bars representing the range of values measured for each network size. The x-axis is shifted downward to show that the convergence times for RSTP with Epochs is negligible compared to those of RSTP. In fact the highest convergence time observed for RSTP with Epochs is only 100 microseconds. This is because RSTP with Epochs does not suffer from the count-to-infinity problem and its convergence is only limited by the inherent network delay. On the other hand, RSTP takes a much longer time to converge. The variance in the convergence times for RSTP is due to the variability in the race conditions when count to infinity occurs.

In the second set of experiments we use simpler "loop" topologies, similar to the topology in Figure 2(a) where we vary the total number of bridges in the loop. For example, a network with 10 bridges means the loop has 9 bridges and



(a) Complete graph topologies. (b) "Loop" topologies. (c) Ring topologies.

Fig. 8. Convergence time in a network of 4 to 10 bridges. In figures (a) and (b) convergence time is measured after the failure of the root bridge. In figure (c) convergence is measured after failure of a link connected to the root bridge. Each experiment is run 100 times and the range of convergence times is shown for the 3 topologies.

the loop is connected to the root bridge that does not lie on the loop. Like in the previous experiment we kill the root bridge and measure the convergence time for both protocols. Figure 8(b) shows the convergence times measured. Again, RSTP with Epochs can converge in at most 400 microseconds in these experiments, but RSTP takes seconds to converge even under this simple network setting.

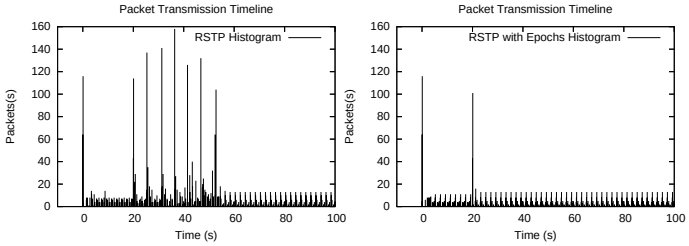
In the third set of experiments we use simple "ring" topologies where the bridges form a simple cycle. We take down the link connecting the root bridge bridge R to a neighbor bridge N. In RSTP, since N does not have any alternate ports, it will declare itself as root and broadcast its BPDU. The BPDU will flow through its descendants, invalidating the topology information at their root ports, until it reaches a bridge with an alternate port to the root. Since the alternate port caches better information, the bridge will pick the alternate port as its root port and will send this new information back to N so it will eventually know that R is alive and accept it as its root. This means that N's BPDU will travel half way around the ring to reach the bridge with the alternate port, then the bridge with the alternate port will send a BPDU that will travel back to N, until N knows that R is alive.

Conversely in RSTP with Epochs, N will detect disconnection from the root, so it will send a BPDU with a higher sequence number than the last BPDU it has received from the root R. This will signal a new epoch to all bridges in the ring and they will accept N's BPDU as it has a higher sequence number. Eventually N's BPDU will reach R after traveling all the way around the loop. R, knowing it is the legitimate root, will in response increase its sequence number and send a new BPDU to assert itself as the root. R's BPDU with the higher sequence number will make its way to N after traveling all the way back around the network. At this point, N will accept R as its root.

The effect of these different behaviors can be observed in Figure 8(c) where RSTP with Epochs takes roughly twice the amount of time to converge compared to RSTP. Note that the convergence times for both protocols are very small in this set of experiments. In these experiments there is no variance in the results as there are no race conditions and thus the results are deterministic.

B. Comparing BPDU Overhead of Both Protocols

In this set of experiments, we present histograms plotting the total number of BPDU packets transmitted in the network within every tenth of a second for both RSTP and RSTP with Epochs using the three families of topologies as used in Section VII-A. We exclude the BPDUs transmitted to or from the root bridge as the root bridge dies at time 20. Thus, we want to factor out the fact of having a different number of BPDUs transmitted in the network before and after the death of the root bridge. Each histogram presents the packet transmissions in the network in a single experiment run.

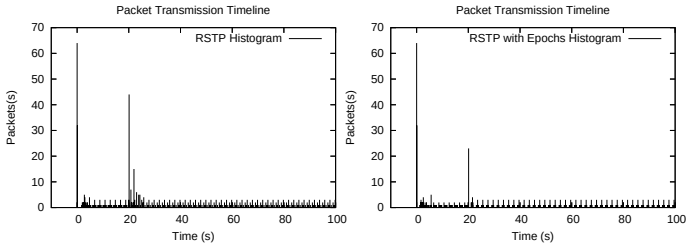


(a) RSTP. (b) RSTP with Epochs.

Fig. 9. Histogram of BPDU packet transmissions in a 10 bridge fully connected graph topology, each bin is 0.1 second. The root bridge dies at time 20.

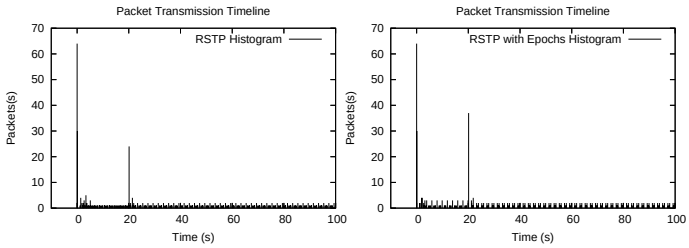
In the first experiment we simulate a complete graph of 10 nodes. We kill the root bridge at time 20. Figures 9(a) and 9(b) show the histograms of BPDUs transmitted for RSTP and RSTP with Epochs respectively during a 100 second time span. For both protocols we observe a spike in the BPDUs transmitted at startup time. This is because at startup each bridge sends out its BPDU and keeps sending out any new better topology information it receives until the bridges in the network agree on the same root and converge to the final spanning tree. After that the network goes into steady state where bridges only send the periodic hello message every HelloTime. At time 20, when the root bridge dies, the two protocols start behaving differently. RSTP suffers from the count-to-infinity problem and sends out a lot of BPDUs during a time span that exceeds 30 seconds until the network converges. RSTP with Epochs reacts differently to the failure of the root. There is an initial spike in the BPDUs transmitted as the new topology information – of the death of the root and a new bridge asserting itself as the new root – flows throughout the network. Then the network converges almost instantaneously and BPDU transmission returns to steady state.

In the second experiment we simulate a topology similar to that in Figure 2(a) with 10 bridges, 9 of them are in the loop. We kill the root bridge at time 20. Figures 10(a) and 10(b) show the histograms of BPDUs transmitted for RSTP and RSTP with Epochs respectively during a 100 second time span. Again, for both protocols we observe a spike in the BPDUs



(a) RSTP. (b) RSTP with Epochs.

Fig. 10. Histogram of BPDU packet transmissions in a 10 bridge "loop" topology, each bin is 0.1 second. The root bridge dies at time 20.



(a) RSTP. (b) RSTP with Epochs.

Fig. 11. Histogram of BPDU packet transmissions in a 10 bridge ring topology, each bin is 0.1 second. A link connecting the root bridge to a neighbor dies at time 20.

transmitted at startup time. After that the network goes into steady state where bridges only send the periodic hello message every HelloTime. At time 20, when the root bridge dies, the two protocols start behaving differently. Similar to the first experiment, RSTP suffers from the count-to-infinity problem and sends out a lot of BPDUs until the network converges. RSTP with Epochs converges almost instantaneously requiring much fewer BPDUs to converge.

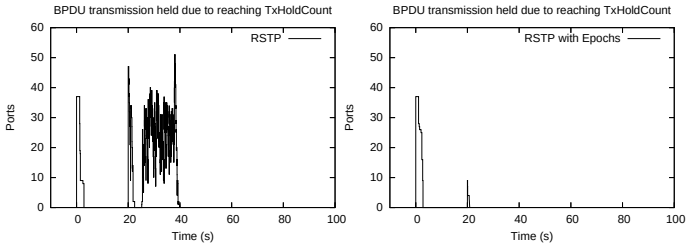
In the third experiment we simulate a 10 bridge ring topology. Similarly, we kill the link connecting the root bridge to a neighbor at time 20. Figures 11(a) and 11(b) show the histograms of BPDUs transmitted for RSTP and RSTP with Epochs respectively during a 100 second time span. In this experiment we observe that RSTP with Epochs uses more BPDUs than RSTP to recover from the failure. This is because as explained in Section VII-A, in RSTP with Epochs the disconnected bridge sends a BPDU that traverses more hops than in the case of RSTP.

In the three sets of experiments we note a short period of time after convergence where there is higher rate of BPDUs being transmitted. This is because of the topology change events that result in an extra BPDU getting transmitted through each bridge's root port every HelloTime and this lasts throughout the duration of the topology change timer.

C. Effect of Count to Infinity on Port Saturation

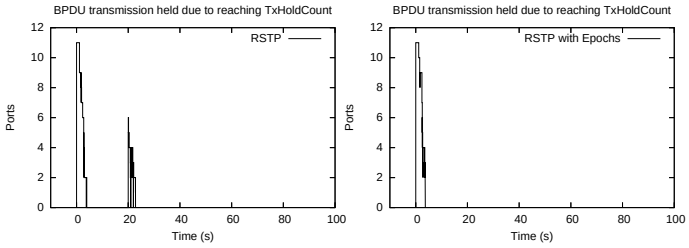
A port is said to be saturated if it has reached its TxHoldCount limit but still has more BPDUs to transmit. We present a time sequence of the number of saturated ports in the whole network in the three experiment scenarios presented in Section VII-B.

In the first experiment simulating a complete graph of 10 nodes we observe in Figure 12 a spike in the number of saturated ports at startup due to the spike in transmitted BPDUs at startup by both protocols. However starting from time 20 when the root port dies, we find a long period of



(a) RSTP. (b) RSTP with Epochs.

Fig. 12. Time sequence of number of ports that have reached their TxHoldCount limit while still having more BPDUs waiting for transmission. This experiment is for a 10 bridge fully connected graph topology where the root bridge dies at time 20.



(a) RSTP. (b) RSTP with Epochs.

Fig. 13. Time sequence of number of ports that have reached their TxHoldCount limit while they still have more BPDUs waiting for transmission. This experiment is for a 10 bridge "loop" topology. The root bridge dies at time 20.

time that is close to 20 seconds in RSTP where the network has many saturated ports. This is due to the count-to-infinity problem where BPDUs spin around the loop causing the ports to quickly reach their TxHoldCount limit. RSTP with Epochs does not suffer from the count-to-infinity problem, thus the ports do not get saturated after the failure.

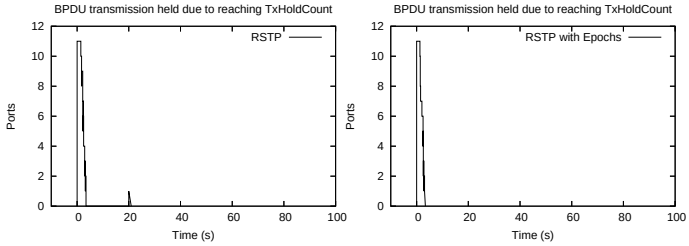
Similarly, in the second experiment – simulating a topology like that in Figure 2(a) with 10 bridges – we observe in Figure 13 a spike in the number of saturated ports at startup. We also observe in RSTP a period after the failure of the root bridge where there are several saturated ports. Again this is because of the count-to-infinity problem.

In the third experiment simulating a ring topology, failure of the root cuts the loop so there is no count to infinity. Thus, for both protocols virtually no ports get saturated after the failure as can be seen in Figure 14.

VIII. Related Work

This paper is based on Elmeleegy *et al.* [4] and contains significant revisions and extensions. While the previous paper characterized the conditions under which count to infinity occurs and observed slow convergence, this paper adds an in-depth explanation for the long convergence time (Section V). While the previous paper only observed that a forwarding loop can be formed during count to infinity, this paper explains the harmful race condition between the RSTP state machines (Section IV-A) and demonstrates that the race condition allows a temporary forwarding loop to be formed (Section IV-B).

The count to infinity behavior of RSTP was mentioned in [11]. However, prior to our work the cause and implications of this problem were neither documented nor understood. Some pathological causes for forwarding loops unrelated to count to infinity have been previously documented by Cisco [1]. However, we are the first to show that count to infinity can lead to a forwarding loop.



(a) RSTP. (b) RSTP with Epochs.

Fig. 14. Time sequence of number of ports reaching their TxHoldCount limit while still having more BPDUs waiting for transmission. This experiment is for a 10 bridge ring topology where a link connecting the root bridge to a neighbor dies at time 20.

To mitigate the limitations associated with using spanning tree for packet forwarding, Perlman proposed Rbridges [13], which employ link state routing among Rbridges rather than relying on spanning trees. Rbridges mitigate the effects of routing loops by encapsulating each packet with an Rbridges header that includes a time-to-live field. Garcia *et al.* proposed replacing the spanning tree with link state routing as well [6], however they do not provide a mechanism to deal with temporary routing loops. SmartBridge [15] uses complex internodal coordination mechanism, namely diffusing computations [2], to achieve effective global consistency and consequently avoid loops. SmartBridges freeze the network and discard all the data during convergence time after a topology change.

STP and RSTP implement a variant of Distance Vector (*DV*) routing. RSTP with Epochs extends RSTP to eliminate the count-to-infinity problem. Other variants of DV routing that are loop-free have been proposed in the literature. For example, Ray *et al.* proposed distributed path computation with intermediate variables (*DIV*) [14], which eliminates loops and prevents count to infinity under distance-vector path computation algorithms. Garcia-Lunes-Aceves [7], Merlin *et al.* [10], and Jaffe *et al.* [8] employ diffusing computations as well when they make modifications to their routing tables to guarantee that their modifications are correct. Perkins *et al.* proposed Destination-Sequenced Distance-Vector (*DSDV*) [12], where every node in the network periodically advertises a monotonically increasing sequence number. The latest sequence number received from a destination is included in its route information in the routing table. A route with a higher sequence number is always preferred over another route to the same destination with a lower sequence number. This is similar to RSTP with Epochs except that in RSTP with Epochs there is only one sequence number that is modified by the root bridge. If the root bridge retires, the sequence number is inherited by the new root bridge. Also RSTP with Epochs only considers sequence numbers across the boundary of two epochs. Within the same epoch, sequence numbers are not considered.

Finally, in order to maintain backward compatibility with deployed Ethernet standards, Elmeleegy *et al.* have proposed the EtherFuse [5], which is a network device that suppresses the harmful effects of count to infinity and forwarding loops in existing Ethernet implementations. However, since the EtherFuse does not actually solve the underlying problems in the spanning tree protocols, spanning tree convergence can still take a long time.

IX. Conclusions

The dependability of Ethernet heavily relies on the ability of RSTP to quickly recompute a cycle-free active forwarding topology upon a partial network failure. In studying RSTP under network failures, we find that it can exhibit a count-to-infinity problem. In our experiments, we show that in some scenarios the count to infinity can extend the convergence time to reach 50 seconds. During the count to infinity, bridges transmit a lot more BPDUs than during its normal operation. Those extra BPDUs cause bridge ports to reach their transmission rate limit, which contributes to extending the convergence time. In this paper, we characterize the exact conditions under which the count-to-infinity problem manifests itself. Then, we show that protocol parameter tuning cannot adequately improve RSTP's convergence time. Also, we uncover race conditions between the state machines in the RSTP specification. Those race conditions when compounded with the count to infinity can cause a temporary forwarding loop. In practice, the likelihood of the formation of the forwarding loop is implementation dependent. However, the consequences of the forwarding loop are so severe that the specification should be corrected. Finally, we propose a simple yet effective solution, RSTP with Epochs. We show that RSTP with Epochs eliminates the count-to-infinity problem and dramatically improves the convergence time of the spanning tree computation upon failure to at most one round-trip time across the network. This solution can therefore significantly enhance the dependability of Ethernet networks.

Appendix

While reading the following proofs we expect the reader to have the IEEE 802.1D (2004) specification at hand. In our proofs, we use the same notation used in Section IV-B. We also use `T` and `F` to refer to values "True" and "False" respectively.

Proof of Claim 3(a): Suppose bridge 1 in Figure 2 dies, causing the start of a count to infinity. In addition, suppose bridge 2 is currently bridge 3's parent, and bridge 2 proposes topology information to bridge 3 that is worse than the information currently at bridge 3 but this does not result in a change of the root port of bridge 3. The following sequence of events shows how an agreement can be sent by bridge 3 in response to the proposal without bridge 3 performing a sync operation.

First, the PORT INFORMATION state machine *(Clause 17.27)* is run on 3p2 when the new information with the proposal from bridge 2 is received in a BPDU. The information is worse than the port priority vector, but it is `SuperiorDesignatedInfo`. The relevant outcomes for 3p2 are: `reselect=T, selected=F and agree=F`.

Second, the PORT ROLE SELECTION state machine *(Clause 17.28)* must be run next, and the relevant outcomes for 3p2 are: `reselect=F, selected=T, agree=F and updtInfo=F`; the relevant outcome for 3p4 is: `updtInfo=T`.

Third, two possible executions can happen depending on which of the two state machines runs next: (a) run the PORT INFORMATION state machine on 3p4, or (b) run the PORT

ROLE TRANSITION state machine *(Clause 17.29)* on 3p2. Suppose (b) runs first. Because the `synced` flag for 3p4 is only reset in the PORT INFORMATION state machine when (a) runs, running (b) first allows `(allSynced && !agree)` and `(proposed && !agree)` to both be true. Thus, this nondeterminism in the PORT ROLE TRANSITION state machine allows it to enter the `ROOT_AGREED` state, instead of the presumedly intended transition into the `ROOT_PROPOSED` state. The relevant outcome from this transition is that `agree=T` at 2p3. Thus, an agreement can be sent to bridge 2 immediately *(Clause 17.21.20)*. Moreover, `setSyncTree()` never gets executed, so the `sync` flag remains false for 3p4. Now (a) runs and the `UPDATE` state is entered. The relevant outcomes for 3p4 are: `agreed=F, synced=F and updtInfo=F`.

Fourth, since `(selected && !updtInfo)` is true for 3p4, the PORT ROLE TRANSITION state machine runs for 3p4. Since (b) was run first and the transition to `ROOT_AGREED` is taken instead of the transition to `ROOT_PROPOSED`, `setSyncTree()` never executes. Consequently, `sync` remains false for 3p4. This means none of the transitions in the PORT ROLE TRANSITION state machine for 3p4 can be taken. The machine does nothing interesting. In particular, it does not transition to `DESIGNATED_DISCARD` as presumedly intended because the `sync` flag is false. Note that the `synced` flag at 3p4 gets set to true as soon as bridge 3 receives a BPDU with the agreement flag from bridge 4. ■

Proof of Claim 3(b): We will again provide an existential proof similar to that given for Claim 3(a). Suppose that bridge 1 dies, causing the start of a count to infinity. In addition, suppose bridge 2 is currently bridge 3's parent, and bridge 2 transmits worse topology information than what bridge 3 currently has. Also suppose that this information is transmitted *without* a proposal to bridge 3 and this information does not result in a change of the root port. The following sequence of events shows how an agreement can be sent by bridge 3 in the absence of a proposal.

The first two events are identical to the first two steps from the proof of Claim 3(a).

Now, two possible executions can happen depending on which of the two state machines runs next: (a) run the PORT INFORMATION state machine on 3p4, or (b) run the PORT ROLE TRANSITION state machine on 3p2. Suppose (b) runs first. Because the `synced` flag for 3p4 is only reset in the PORT INFORMATION state machine when (a) runs, running (b) first allows `(allSynced && !agree)` to be true. Thus, the PORT ROLE TRANSITION state machine enters the `ROOT_AGREED` state. The relevant outcome for this transition is that `agree=T` at 2p3. Thus, an agreement can be sent to bridge 2 immediately *(Clause 17.21.20)*. Now (a) runs and the `UPDATE` state is entered. The relevant outcomes for 3p4 are: `agreed=F, synced=F and updtInfo=F`. Also note that the `synced` flag at 3p4 gets set to true as soon as bridge 3 receives a BPDU with the agreement flag from bridge 4. ■

REFERENCES

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

[2] E. W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):14, August 1980.

[3] K. Elmeleegy. RSTP with Epochs Simulator. http://www.cs.rice.edu/~kdiaa/ethernet/, 2007.

[4] 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.

[5] K. Elmeleegy, A. L. Cox, and T. S. E. Ng. EtherFuse: An Ethernet Watchdog. In *ACM SIGCOMM 2007*, Aug. 2007.

[6] 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.

[7] J. J. Garcia-Lunes-Aceves. Loop-Free Routing Using Diffusing Computations. *IEEE/ACM Transactions on Networking*, 1(1):130–141, February 1993.

[8] J. M. Jaffe and F. H. Moss. A Responsive Distributed Routing Algorithm for Computer Networks. *IEEE Transactions on Communications*, 30:1758–1762, July 1982.

[9] 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.

[10] P. M. Merlin and A. Segall. A Failsafe Distributed Routing Protocol. *IEEE/ACM Transactions on Networking*, 27:1280–1287, September 1979.

[11] 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.

[12] C. E. Perkins and P. Bhagwat. Highly dynamic Destination-Sequenced Distance-Vector routing (DSDV) for mobile computers. In *ACM SIGCOMM 1994*, pages 234–244, Aug. 1994.

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

[14] S. Ray, R. A. Guerin, and R. Sofia. Distributed Path Computation without Transient Loops: An Intermediate Variables Approach. In *20th International Teletraffic Congress Plenary (ITC 20)*, June 2007.

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

Khaled Elmeleegy received his B.S. degree in Computer Engineering with distinction and honors from the Faculty of Engineering, Alexandria University, Egypt in 1998 and M.S. degree in Computer Science from Rice University, Houston, TX in 2004.

He is currently a Ph.D. candidate in Computer Science at Rice University, Houston, TX. His research interests include network infrastructure reliability, high performance network servers, operating and distributed systems.

Alan L. Cox is an Associate Professor of Computer Science at Rice University, Houston, TX, where he received the NSF Young investigator (NYI) Award in 1994 and a Sloan Research Fellowship in 1998. Previously, he earned the B.S. degree in applied mathematics from Carnegie Mellon University, Pittsburgh, PA, in 1986 and the M.S. and Ph.D degrees in Computer Science from the University of Rochester, Rochester, NY, in 1988 and 1992, respectively. His recent research has focused on network I/O virtualization in virtual execution environments, performance analysis tools for distributed applications, and network infrastructure reliability.

T. S. Eugene Ng is an Assistant Professor of Computer Science at Rice University. He is a recipient of a NSF CAREER Award in 2005. He received a B.S. in Computer Engineering with distinction and magna cum laude in 1996 from University of Washington, a M.S. in Computer Science in 1998 and a Ph.D. in Computer Science in 2003 from Carnegie Mellon University. His research interest lies in developing new network models, network architectures, and holistic networked systems that enable a robust and manageable network infrastructure.