Supplemental Note on Count-to-Infinity Induced Forwarding Loops in Ethernet Networks

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

Abstract— **Ethernet forwarding loops are dangerous. Packets can be trapped in a loop and cause network congestion and instability. Furthermore, packet forwarding can be disrupted due to the pollution of forwarding tables. In this report, we show that the "count-to-infinity" behavior in Ethernet's Rapid Spanning Tree Protocol (RSTP) can lead to a temporary forwarding loop. Additionally, we identify the causes for the formation of this forwarding loop: races between protocol control packets traversing the network, races between RSTP state machines, and nondeterminism within RSTP state machines. Finally, we present an annotated trace of RSTP events for an example network as an existential proof of the formation of a forwarding loop during count to infinity.**

I. Introduction

Myers *et al.* [3] observed that Ethernet's distributed forwarding topology computation protocol – the Rapid Spanning Tree Protocol (RSTP) [2] – can suffer from a classic "count-to-infinity" problem. Elmeleegy *et al.* [1] explained the origin of this problem in detail, studied its behavior under a wide range of scenarios, and proposed a simple solution, RSTP with Epochs, that addresses the count to infinity problem.

In addition, Elmeleegy *et al.* observed that a forwarding loop can be formed during count to infinity. This report supplements Elmeleegy *et al.* and provides a detailed explanation of how a forwarding loop can form during count to infinity in RSTP.

Such forwarding loops are serious problems. During the count to infinity, which can last tens of seconds even in a small network, a forwarding loop can cause a network to become highly congested by packets that persist in the loop. Moreover, packet forwarding can fail due to the pollution of forwarding tables with false information that is learned from looping packets.

There are three key ingredients for the formation of a forwarding loop during count to infinity:

1) Count to infinity is initiated around a physical network cycle. This behavior and its cause are described by Elmeleegy *et al.* [1]. This results in the propagation of BPDUs carrying both fresh and stale information around the network cycle.
2) During count to infinity, the fresh information stalls at a bridge because the bridge port has reached its TxHoldCount and subsequently the stale information is received at the bridge. As as 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.
3) The "sync" hand-shake operation that would have prevented a forwarding loop is not performed by a bridge when a proposal BPDU carrying worse information is received at its root port, allowing the forwarding loop to be formed. This behavior is a result of race conditions that exist between the RSTP state machines and nondeterministic transitions within state machines. These race conditions and nondeterministic transitions have not been previously documented. Explaining them is a key contribution of this report.

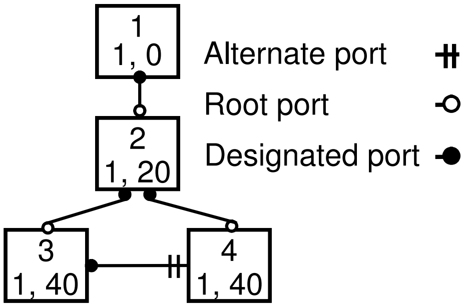

Fig. 1. An example network of bridges.

The rest of this report explains these problems in detail. First, Section II explains the aforementioned race conditions and nondeterministic transitions in RSTP in detail. Then, Section III provides 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.

II. Race Conditions

In this section we describe the race conditions between the RSTP state machines that allow for the formation of the forwarding loop. The RSTP state machines can be found in the 802.1D (2004) specification [2]. Those 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 races naturally occur between the RSTP state machines and some of them can be harmful.

We will use the example network as shown in Figure 1 to illustrate how the races play out. A box represents a bridge; the top number in the box is the bridge ID, the lower set of numbers represent the root bridge ID as perceived by the current bridge and the cost to this root. The link costs are all 20. This value is not important though.

In this section and the rest of this report we use the notation ipj to name a port of a bridge, where ipj means the port at bridge i connecting it to bridge j. We use a fixed-width font to refer to state machine variables, all capital letters to refer to state machine names, and a fixed-width font with all capital letters to refer to state names.

We present two lemmas that describe the races and prove their existence. The two lemmas are similar. Lemma 1 differs from Lemma 2 in that it handles the case where the bridge in question receives the new information along *with a proposal* as explained later in the section.

Lemma 1: Race conditions exist if a bridge receives, from its designated bridge at its root port, worse information than what it currently has and this received information comes along *with a proposal* that would not result in the bridge changing its root port. These race conditions can cause the bridge to respond with an agreement to its designated bridge without doing a sync operation.

Proof: Consider if bridge 1 in Figure 1 dies, causing the network of bridges to start counting to infinity. Suppose bridge 2 is currently bridge 3's designated bridge, and bridge 2 proposes 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.

1) 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`.
2) 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`.
3) 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 *(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 presumed 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. 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`.
4) Now, since `(selected && !updtInfo)` is true for 3p4, the PORT ROLE TRANSITION state machine gets run for 3p4.

   Since (b) was run first, and the `ROOT_AGREED` state is taken instead of the `ROOT_PROPOSED` state, `setSyncTree()` never got executed, and so `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 presumed 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.

■

Lemma 2: Race conditions exist if a bridge receives, from its designated bridge at its root port, worse information than what it currently has and this received information comes along *without a proposal* that would not result in the bridge changing its root port. These race conditions can cause the bridge to respond with an agreement to its designated bridge.

Proof: We will again provide an existential proof similar to that given in Lemma 1. Consider that bridge 1 dies, causing the network of bridges to start counting to infinity. Suppose bridge 2 is currently bridge 3's designated bridge, and bridge 2 transmits worse 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.

1) The first two events are identical to (1) and (2) from the proof of Lemma 1.
2) 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.

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.

■

III. 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 a forwarding loop. Table I shows a trace of events after the failure of the root bridge, bridge 1, in the network shown in Figure 1. The first column of the table shows the time of occurrence for each event in increasing order. The second column is used if the event is a BPDU transmission, where it shows the sender and receiver bridges of the BPDU. The third column shows the contents of the BPDU if this is a BPDU transmission event. The fourth column shows additional comments describing the event.

Assume that bridge 1 has died right after bridge 2 has sent out a Hello message but before its clock has ticked. Thus, the Transmission Count (TxCount) is one for ports 2p3 and 2p4 and zero for ports 3p2, 3p4, 4p2, and 4p3.

Also assume that bridges use a Transmission Hold Count (TxHoldCount) value of 3. Thus, each port can transmit at most 3 BPDUs per clock tick. 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.

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. *(Lemma 2)*

At t4, bridge 3 will pass the information it received from bridge 2 to bridge 4.

Bridge 4 having a cached path at its alternate port to the retired root, bridge 1, believes 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 synchronization operation (sync) 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 [2]. 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 designated bridge.

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 gone 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.

At t10, the information from bridge 3 announcing bridge 2 to be the root arrives at bridge 4, bridge 4 then passes on 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. *(Lemma 2)*

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 designated bridge.

At t14, bridge 2 responds to bridge 4's proposal with an agreement, notifying bridge 4 that it agrees to bridge 4 being its designated bridge. 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 [2].

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 on the information to bridge 3 nor send back a response to the proposal coming along with the new information from bridge 4. This is because both 2p4 and 2p3 have reached their TxHoldCount limit preventing them from sending any BPDUs during this clock tick. The fresh information that conveys bridge 2 should be the root is stalled at bridge 2 as a result.

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 [2].

Time	BPDU Direction	BPDU Contents (Root, Cost[, Flags])	Comments
t1	B2 → B3	2, 0	
t2	B2 → B4	2, 0	
t3	B3 → B2	2, 20, Agreement	
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	
t10	B4 → B2	2, 40, Topology Change, Proposal	
t11	B4 → B3	2, 40, Topology Change, Agreement	
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)*
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)*
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.
t22	B4 → B2	1, 120, Topology Change, Proposal	Occurs after a clock tick at B4 decrementing TxCount.
t23			Reroot at B2, 2p4 is the new root port; sync, 2p3 is already blocked.
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.
t26			Unblock 4p2, agreement arrives.
t27	B3 → B2	1, 160, Topology Change, Agreement	
t28	B3 → B4	1, 160, Topology Change	
t29			Unblock 2p3, agreement arrives.

TABLE I

AN EXAMPLE SEQUENCE OF EVENTS, AFTER FAILURE OF THE ROOT BRIDGE IN FIGURE 1, THAT LEADS TO A FORWARDING LOOP.

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.

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.

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 designated bridge 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.

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.

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 *(Lemma 1)*.

Bridge 3 also passes on 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.

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 bridge 1 is the root. None of them will carry a proposal flag. No bridge will perform any sync operation. Thus the forwarding loop will persist until the count to infinity ends when the stale information conveying bridge 1 is the root is aged out.

References

[1] 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.

[2] 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.

[3] 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.