Link state bridging part 2

Mick Seaman

A prior note[1] described a neighbour to neighbour tree agreement protocol (TAP) that ensures loop-free topology(ies) for one or more trees, each calculated by any protocol capable of determining the *Root Identifier*, *Root Port*, and *root path cost*[2] for each bridge. That protocol makes explicit *root path priority vector* promises to the bridge's neighbours: allowing interoperability between different topology protocols (each bridge's promises are comprehensible to its neighbour at all times) and maximizing the potential for connectivity, but with the disadvantage of requiring per tree TAP messages. This note proposes a link state protocol specific TAP design, with the promises for all trees represented by a brief digest of the input to the link state calculation, enhancing scalability by substituting local computation for per tree communication. Both TAP protocols might be used together in bridging applications, with the per tree design just for the CIST to provide plug-and-play interoperability between dynamically determined link state bridging *Regions*.

Concepts and terminology from the prior note are used, without reintroduction. Loop free operation at all times is an absolute requirement—irrespective of the number of link state updates in progress and the order of their arrival (or temporary non-arrival) and inclusion in the link state computation at each bridge. If neighbouring bridges have a completely different recent update history they will be unable to draw conclusions from each other's link state digests, and thus unable to create new safe connectivity, but the design enables rapid *Root Port* failover[3] without any protocol exchanges in (at least) the common case of a single link change. As an improvement over simply *cutting* the active topology whenever neighbours' link state differs, this is particularly effective in structured networks; and essential if link state bridging is always to perform at least as well[4] as RSTP/MSTP.

Each bridge performs the link state calculation for itself and its immediate neighbours to determine[5] its *Port Roles* for each tree; and records the value of the Contract that would be implied by the transmission of that digest on each *Designated Port* and by its potential receipt on the *Root Port* and each of the *Alternate Ports*. A *Root Port* is *Forwarding* if it holds a received Contract and has not transmitted an outstanding Contract. A *Designated Port* is made *Discarding*, prior to a newly calculated *Root Port* becoming *Forwarding*, unless it has no outstanding Agreements and has received an Agreement for one of its outstanding or potential promises with the value of the matching Contract and all outstanding Contracts on the port being *worse* than that of the *worst* outstanding potential Contracts on the *Root Port*. Otherwise the *Designated Port* is made *Forwarding*, unless the *Root Port* is, and is to remain, *Discarding*. In that case the *Designated Port* is *Forwarding* iff it has no outstanding Agreements and has received an Agreement for one of its outstanding or potential promises.

Prior promises, both outstanding and received, are discarded when a matching promise (digest) is received or calculated. The link state TAP PDUs convey a single digest, attesting to the transmitter's state and its prior application of any necessary filtering arising from outstanding promises. Each participant can thus continue to forward using the last synchronized calculation as a base, possibly adding some forwarding to reflect root port failover for some trees and removing other forwarding to prevent loops, until it and its neighbour agree on the same link state information once more.

---

[1]Link state bridging, Mick Seaman, 24th March 2008.

[2]All italicised terms are defined by IEEE Stds 802.1D and/or 802.1Q.

[3]See IEEE Std 802.1Q clause 13.6, and High Availability Spanning Tree, Mick Seaman, October 1998, see also A Framework for Loop-free Convergence, M. Shand, S. Bryant, February 2008, section 5 (destinations of type A2, with asymmetric cost criteria).

[4]Successful introduction of an 'improved' protocol usually requires that it performs at least as well as that it seeks to better, particularly in common cases. This note was prompted, in part, by a suggestion by Don Fedyk and others that used a link state digest but did not allow for protocol-less failover.

[5]Efficiencies are believed to be possible, so this does not add an amount of work equivalent to that for a single calculation times the number of neighbours.

1. Summary

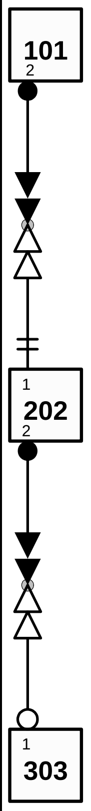
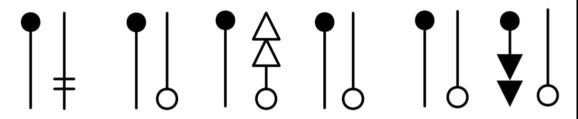
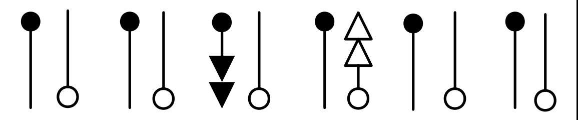
This note begins by considering the protocol interactions between three connected bridges, following a single link state change, with a particular emphasis on Root Port failover (section 2). Section 3 considers multiple changes: initially all bridges agree on the input to the link state calculation (as is evidenced by their agreement on the digest), but during the course of updates each undertakes calculations whose digest is not (and is never) comprehensible to the others. Finally all agree on a new digest, and convergence on the new topology is complete.

Section 4 considers aspects of the link state TAP design, in particular how many digests should be allowed to be outstanding or should be held on receipt.

2. A single link state change

Consider three bridges, 101, 202, and 303 (say), connected by point-to-point links (101 to 202, 202 to 303). Let *a*, *b*, *c*, .. represent the digests that reflect states of the link state calculation input, e.g. inclusion (or not) of particular LSPs. Table 1 shows each bridge and bridge ports' state (for some of the trees).

Table 1—Promises and forwarding with a single change

	Step	0	1	2	3	4	5
	101 State (digest)	*a*	*a*	*a*	*a*	*b*	*b*
	101.2 Outstanding Promises	*a*	*a*	*a*	*a*	*b*	*b*
	101.2 Received Promises	*a*	*a*	*a,b*	*a,b*	*b*	*b*
	101.2 Forwarding	*a*	*a*	*a*	*a*	*b*	*b*
							
	202.1 Forwarding	*a*	$b^{a.a}$	$b^{a.a}$	$b^{a.b}$	$b^{a.b}$	*b*
	202.1 Outstanding Promises	*a*	*a*	*a,b*	*a,b*	*a,b*	*b*
	202.1 Received Promises	*a*	*a*	*a*	*a*	*a*	*b*
	202 State (digest)	*a*	*b*	*b*	*b*	*b*	*b*
	202.2 Outstanding Promises	*a*	*a*	*a,b*	*b*	*b*	*b*
	202.2 Received Promises	*a*	*a*	*a*	*b*	*b*	*b*
	202.2 Forwarding	*a*	$b^{a.a}$	$b^{a.a}$	$b^{a.b}$	$b^{a.b}$	*b*
							
	303.1 Forwarding	*a*	*a*	*a*	$b^{b.?}$	*b*	*b*
	303.1 Outstanding Promises	*a*	*a*	*a*	*b*	*b*	*b*
	303.1 Received Promises	*a*	*a*	*a,b*	*b*	*b*	*b*
	303 State (digest)	*a*	*a*	*a*	*b*	*b*	*b*

Initially all three and other connected bridges, have the same link state information, represented by digest *a*. The fraction of the initial active topology supported by the three bridges is the same for all the trees shown: the *Root* is at or above 101, 202's *Root Port* connects to another bridge, and 303's *Root Port* connects directly to 202. The *Port States* shown are just for those trees whose *Root* remains at or above 101, with 202's new *Root Port* (202.1) connecting to 101, and 303's *Root Port* remaining unchanged (see prior note for network reconfiguration examples), but the link state digests and promises apply to all trees.

In this scenario, 202's link state information is updated first (step 1), with new digest *b*, causing 202.1 to be selected as the new Root Port for some trees. None of the bridge's have yet sent further TAP messages, and (if the link state change is a result of 202's local detection of the prior *Root Port's* link failure) it is possible that no protocol exchanges have taken place at all. However 202 can immediately modify its frame Forwarding to include trees whose frames would be forwarded if all nodes were using the link state information identified by *b*, subject to the proviso that the Contract previously received (for that tree) from 101 is *better* than the Contract previously agreed to by 303. This corresponds to the observation previously made for RSTP/MSTP: if a Root Port is blocked, an Alternate Port can be made Forwarding immediately provided that spanning tree information was previously synchronized through the active topology encompassing the new connectivity. The Forwarding state is shown as $b^{a.a}$, i.e. forward using link state information identified by b for each tree where the Root Port's received Contract *a* is better than that agreed to by the Designated Port's peer (*a* again).

In step 2, 202 transmits a fresh Agreement (to 101) and Contract (to 303) for *b*. However 202 does not withdraw its prior Promises, which is just as well, for although 101 and 303 receive and record the Promise for *b*, they have as yet no idea what *b* signifies.

In step 3, 303 has updated its topology with the information for digest *b* (calculating *b* at the same time). 303 knows that *b* is more recent than *a*, so it can discard the received Contract for *a*, and send a TAP message to 202 also promising Agreement *b*, it can discard its outstanding promise for *a* immediately (since 303 knows *b* was more recent than *a*, 202's Agreement for *b* means that it has no further use for *a*—and the protocol (see Section 4 and 5) will ensure that the discard is communicated to 202). Note however that 303 considers its Agreement *a* to be outstanding until it has received a subsequent message from 202 discarding that Agreement. 303 is now forwarding $b^{b.?}$, where the '?' represents the Agreement(s) received at its Designated Port—either '*a*' or '*b*' (note $b^{b.b} == b$). 202's forwarding is now $b^{a.b}$, i.e. forwarding according to trees calculated according to *b*, but omitting forwarding for which 101's Contract *a* is not better than the Contract *b* promised to 303.

In step 4, 101 completes its topology calculation with the information identified by *b*. It too knows that *b* is more recent than *a* (having calculated both it knows that *b* is as *a*, but with the inclusion of link state updates with more recent sequence numbers), and can discard the received promise for *a*. Since it has 202's Agreement for *b*, it can start forwarding using *b* alone, and can also discard its outstanding promise for *a*.

In step 5, 101 sends Contract *b* to 202, withdrawing Contract *a* and discarding Agreement *a* with the same message. On receipt, 202 can remove *a* from its forwarding and promise state.

The steps, and state information at each state, shown in Table 1 is independent of the Port States of each tree, and in this context the distinction between Contracts and Agreements is immaterial and is not explicitly encoded in TAP messages—there is no particular disadvantage in requiring the appropriate forwarding state to be established for all trees, regardless of direction, prior to transmitting promises.

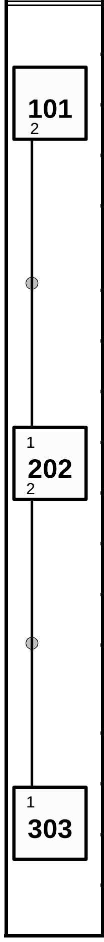
3. Multiple link state changes

Of course Table 1 represents the easy case, a single change, in which a loop will not occur in any case. Table 2 considers multiple simultaneous changes. All bridges start with the same information (digest *a*, step 0), but 101 updates to *b*, 202 to *c*, and 303 first up to *d*, (step 1) before all the bridges reach *e*.

In Table 2 step 2, 202 acquires all the updates (digest *e*) and transmits fresh promises. 202 knows that *a* was intelligible to 101 and 303, since promises for *a* have been received, but does not know their opinion of *c*—it is possible that promises for *c* are in flight: so 202 does not attempt to decide which digests its neighbours want to hold on to. Moreover 202 doesn't know which promises are to be used. Its frame forwarding is represented as $e^{a,c.a,c}$: starting with the forwarding connectivity of *e*, forwarding between a tree's Root Port and each of its Designated Ports is removed if the *worst* of the Root Port's Contract calculated using the information sets *a*, *e*, and *c* is not better than the *best* of the Designated Ports Contracts calculated using the same sets.

However, while 202's promises for *c* and *e* are both (as yet) unintelligible to 101, the latter knows their order of calculation by 202, and that *e* will in time supersede *c* everywhere (until e is itself replaced). So the chance of 202 being able to assign a meaning to *c*, by receiving and processing the necessary link state updates, before receiving and processing those for *e* is small. Accordingly 202 discards the promise for *c* (in step 3), as does 303. Note that 101 does not know if *b* and *e* are strictly ordered, and, if *b* represents more than two changes to *a*, does not know that *b* will not supersede *e*.

Table 2—Promises and forwarding with multiple changes

	Step	0	1	2	3	4	5	6
	101 State (digest)	*a*	*b*	*b*	*b*	*e*	*e*	*e*
	101.2 Outstanding Promises	*a*	*a,b*	*a,b*	*a,b*	*e*	*e*	*e*
	101.2 Received Promises	*a*	*a,c*	*a,e*	*a,e*	*e*	*e*	*e*
	101.2 Forwarding	*a*	$b^{a.a}$	$b^{a.a}$	$b^{a.a}$	*e*	*e*	*e*
			↓ ↑	↑	↓	↓		
	202.1 Forwarding	*a*	$c^{a.a}$	$e^{a,c.a,c}$	$e^{a.a}$	$e^{e.a}$	*e*	*e*
	202.1 Outstanding Promises	*a*	*a,c*	*a,c,e*	*a,e*	*e*	*e*	*e*
	202.1 Received Promises	*a*	*a,b*	*a,b*	*a,b*	*e*	*e*	*e*
	202 State (digest)	*a*	*c*	*e*	*e*	*e*	*e*	*e*
	202.2 Outstanding Promises	*a*	*a,c*	*a,c,e*	*a,e*	*a,e*	*e*	*e*
	202.2 Received Promises	*a*	*a,d*	*a,d*	*a,d*	*a,d*	*e*	*e*
	202.2 Forwarding	*a*	$c^{a.a}$	$e^{a.a}$	$e^{a.a}$	$e^{e.a}$	*e*	*e*
			↓ ↑	↓	↑		↑	↓
	303.1 Forwarding	*a*	$d^{a.a}$	$d^{a.a}$	$d^{a.a}$	$d^{a.a}$	$e^{e.?}$	*e*
	303.1 Outstanding Promises	*a*	*a,d*	*a,d*	*a,d*	*a,d*	*e*	*e*
	303.1 Received Promises	*a*	*a,c*	*a,e*	*a,e*	*a,e*	*e*	*e*
	303 State (digest)	*a*	*d*	*d*	*d*	*d*	*e*	*e*

In step 4, 101 completes a link state computation with digest *e*, and has also received promises for *e* on its other ports. 101 transmits a TAP message promising *e* and discarding the received promise for *a*. It can also discard its outstanding promises for *a* and *b*, since it now knows them to be precursors of *e* (for which it has a received promise). Receipt of 101's promise for *e* allows 202 to discard the received promises for *a* and *b*, and the outstanding promise for a.

In step 5, 303 completes its link state computation with digest *e*, and its change in state, transmission, and and the consequent changes in 202 proceed much as for 101/202 in step 4. Finally in step 6 the other ports of 303 have also received suitable promises, and frame forwarding for all three bridges proceeds using the link state information corresponding to digest *e* alone.

4. Link state TAP design considerations

The fact that all digests are not necessarily intelligible to the recipient of a promise means that the link state digest based TAP differs significantly from that using per tree distance vectors. Each promise does not necessarily replace its predecessor. However each port only needs to retain at most two received promises: the last one whose digest matches a digest computed by the recipient, and the last one received.The question is whether the full state of the transmitter[1] should be included in each message or not. This would require including between one and four digests in each message[2], and would allow two neighbours to sync up on digest that is not necessarily their latest—enabling new connectivity even as a reconfiguration is in progress. On balance the benefit seems marginal[3]—

[1]Usually a sound principle in protocol design, though sometimes requiring the handling of more protocol fields than desirable.
[2]Allowing for a window rotation so that an issued digest can be replaced by its successor
[3]I have a design if it is wanted.

both a loop free guarantee and a rapid protocol-less failover to an Alternate Port can be provided by communicating a single digest. Addition of a single bit indicating if the transmitter has received the same digest value might be worthwhile—as a diagnostic, and allowing more rapid retransmission in the case of message loss. Providing windowing or pacing of new promises would be better done[1] by using multiple digests, and a bridge's forwarding has to take account of all outstanding promises since the last matching digest received from each neighbour. However the latter can be easily achieved (and the requirement for windowing avoided) by recording the *worst* potential received or the *best* potentially issued Contract outstanding for each port and tree[2] since (and including) the last time a matching digest was received, as well as values for the current topology.

5. TAP event handling

The events to be considered in the link state TAP state machine design are:

— Availability of a new link state topology following receipt or generation of an LSP and link state recalculation.

— Transmission of a TAP message on a port

— Receipt of a TAP message on a port.

The processing of each event specified in this section is atomic, and is not interrupted by processing for any of the other events.

5.1 TAP data

The following variable(s) are maintained for the bridge as a whole:

— ownDigest: The value of the digest corresponding to the last completed link state calculation.

The following variables are maintained on each port, for each tree:

— latestRole: Designated, Alternate, Root, or Backup.

— latestContract: A priority vector.

— latestOutstanding: True, or False.

— priorRole: Designated, Alternate, Root, or Backup.

— chgdRole: True, or False.

— priorContract: A priority vector.

— rcvdDigest: The last received digest.

For a *Root*, *Alternate*, or *Backup Port* the value of the Contract is that (that would be) associated with the digest by the port's neighbour. For a *Designated Port* it is the value assigned by this bridge.

Note—chgdRole is initialized False, so a digest match is required before the port will become *Forwarding*.

5.2 New link state topology

Calculation of a new link state topology involves determining not only the Root Port for each tree, but also the value of the *designated priority vector* for the neighbouring bridge attached to that port, and the *designated priority vectors* for each of the other ports and their attached neighbours (assuming in each case that the neighbours are using the same link state database to perform the calculation). This allows the bridge's ports' *Port Role* (*Designated*, *Alternate*, or *Backup*) and the value of the potential promise to be determined for each tree.

The variable ownDigest for the topology is calculated, using the data input to the link state calculation.

— latestOutstanding is set True; and

— latestRole is set to the calculated *Port Role*; and

— latestContract is set to the calculated priority vector.

If ownDigest matches rcvdDigest[3]:

— chgdRole is cleared, i.e. becomes False; and

— priorRole is set to latestRole; and

— priorContract is set to latestContract.

The new (and now current) link state topology is used to assign Port Roles for the tree, and Port States are assigned using the latestRole, priorRole, latestContract, and priorContract variables (see 5.5).

A request for the opportunity to transmit a TAP message is made. This may be granted immediately, or not, depending on such factors as transmission rate limiting, availability of buffers etc. Transmissions may also be scheduled at regular intervals.

5.3 TAP message transmission

An opportunity to transmit a TAP message through a port results in the following processing.

If latestOutstanding is False:

— If latestRole is Designated:
  - If priorRole is not Designated, chgdRole is set True.
  - Otherwise, i.e. if priorRole is Designated, priorContract is set to the *better* of the values of priorContract and latestContract.

— If latestRole is not Designated:
  - If priorRole is Designated, chgdRole is set True.

[1]Safer, less complex protocol, easier to determine what is happening by observing the protocol.

[2]If the same port has potentially both issued and received and received a contract for the same tree through a given port, it has adopted both *Designated Port* and *Root* or *Alternate Port* roles, and must be *Discarding*.

[3]It is handy to have a reserved or illegal value for the digest calculation, that can be used for 'no digest received', otherwise a separate flag should be supplied to cover the case of startup (i.e. before any digest has been received).

- Otherwise, i.e. if priorRole is not Designated, priorContract is set to the *worse* of the values of priorContract and latestContract.

— The new (and now current) link state topology is used to assign Port Roles for the tree, and Port States are assigned using the latestRole, priorRole, latestContract, and priorContract variables (see 5.5).

NOTE—If the above condition is satisfied when a new link state topology is available (5.2), and an opportunity to transmit is expected immediately, the Port States can be updated just once.

A TAP message is used to encode and transmit ownDigest, and latestOutstanding is set True.

5.4 TAP message receipt

If the digest received in the TAP message does not differ from the previously rcvdDigest, the received message is discarded and requires no further processing. Otherwise rcvdDigest is updated with the received value, and if it now equals ownDigest:

— chgdRole is cleared, i.e. becomes False; and
— priorRole is set to latestRole; and
— priorContract is set to latestContract; and
— latestOutstanding is set True; and
— The new (and now current) link state topology is used to assign Port Roles for the tree, and Port States are assigned using the latestRole, priorRole, latestContract, and priorContract variables (see 5.5).

5.5 Port State assignment

A *Root Port* is made *Forwarding* if chgdRole is False. If the *Root Port* is *Forwarding*, each *Designated Port* whose chgdRole is False and whose priorContract is *worse* than the *Root Port's* priorContract is made *Forwarding*. All other ports are made *Discarding*.

6. Further thoughts

Since a link state digest only makes sense to another bridge with exactly the same link state information, the use of a digest rather than an explicit per tree priority vector could reduce the opportunities to restore communication during reconfiguration. However that opportunity comes at the expense of more communication, and it is only communication delays that prevent two bridges having the same link state information. So in general the difference is likely to be arguable, if not insignificant, on this point.

More significantly perhaps, there is no way of delaying cutting on a per tree basis when using a link state digest to summarize the state for all trees, as there is no way (without some per tree communication) to indicate that the tree has reached root-ward agreement or leaf-ward agreement while continuing communication for some trees. The problem is that there is no place to start such an agreement propagation from in general—at the leaves of one tree we find ourselves in the middle of other trees. This means that communication can be interrupted when per tree agreement would allow a branch to drop (in priority space) as a whole and postpone cuts until requested (by a Proposal) so that new connectivity could be made elsewhere.

Fast distribution of link state updates, as suggested in the prior note, would probably make more difference to overall reconfiguration times than complicating TAP. If the CIST alone were to use per tree TAP, that should make it maximally available, and if both ends of a failed link can send the update before the two parts of the CIST adjust that should ensure that all bridges receive rapid notification of a single failure.

This note has concentrated on the advantages of providing rapid Root Port failover (for example Figures see the prior note). This is important in well designed structured networks (e.g. on campus). In simple rings the alternative proposed by Don Fedyk et al. can be expected to do as well, and possibly slightly better. Both approaches require a single message to be sent by each node on each link (for a single recalculation) but the more aggressive cutting in the alternative means that only one message needs to be received (through the new *Root Port*) to start *Forwarding* towards the *Root*, and in a simple ring repair after a failure either leaves the path to the Root completely unchanged or causes a tree 'flip'.

The description in this note deals with the general case of bi-directional general trees where the destination (or source) of the traffic is not at the root of the tree. It can be trivially extended to the latter case, and to the particular optimization of the latter case when it is only forwarding through the Root Port that is enabled or disabled—but that has not been done yet.

The approach taken in this note is to adopt very simple rules for avoiding loops, with those only involving local checking. A more complex analysis of the entire topology could find more loop-free paths for immediate use after a failure, but the requirement that an arbitrary number of changes can be in progress at any one time is a tough one. I have avoided using any method that would require a worst case fall back to a method that provided an in sync base-line for the entire network.