Annexes

Annex A.

(informative)

Bibliography

The following publications are recommended as background material for understanding the objectives behind this standard:

[B1] IEEE Std 1596-1992, Scalable Coherent Interface.[1]

[B2] IEEE Std 1394-1995, High Performance Serial Bus.[4]

Annex B.

(normative)

Transmit clock synchronization

[1] ANSI/IEEE publications are available from the Institute of Electrical and Electronics Engineers, Service Center, 445 Hoes Lane, P. O. Box 1331, Piscataway, NJ 08855-1331, USA.

Annex C.

(normative)

10G Ethernet PHY

C.1 ...

C.2 ...

C.2.1 ...

C.2.1.1 ...

C.2.1.2 ...

C.2.1.3 ...

C.2.1.4 GMII structure

Figure C.1—Gibabit ethernet reconciliation sublayer (GERS)

The data-transmission path provides the following signals:
TXD<7:0> are data-byte bits: the 0'th through 7'th bit are the first through last transmitted data bits.
TXx_EN is one when a data byte is transmitted; this bits stays asserted during each frame transmission.
TX_ER is an error indication, that is asserted to abort data-frame transmissions (TBD).
COL (collision detected) is not used within this RPR standard.

The data-reception path provides the following signals:
RXD<7:0> are data-byte bits: the 0'th through 7'th bit are the first through last received data bits.
RX_EN is one when a data byte is transmitted; this bits stays asserted during each frame reception.
RX_ER is an error indication, that is asserted to abort data-frame receptions (TBD).
(Please discuss when this is asserted—is it asserted after a bad CRC is detected?).
CRS (carrier sense) is not used within this RPR standard.

PHY_READY.indicate is a control signal that is asserted when another request can be accepted.
PHY_DATA.request represents the request frame contents.

PHY_DATA.indicate is ???.
PHY_DATA_VALID.request is ???.
PHY_LINK_STATUS.indicate is ???.

C.3 ...

C.3.1

C.3.1.1

C.3.1.2

C.3.1.3

C.3.1.4 XGMII structure

The mapping of MAC-supplied data values to transmitted GERS signals is illustrated in Figure C.2. The interface numbering convention is to apply the 0-to-7 bit labels to the least- through most-significant bits within each transmitted data byte. The correspondence of TXC<3:0> signals to associated bytes is specified at the bottom of Figure C.2.

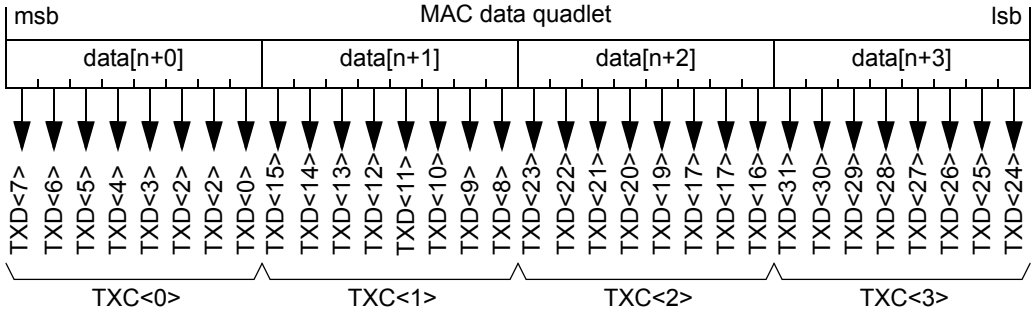


Figure C.2—XGERS transmit signal mapping

The mapping of received GERS signals to MAC-provided data values is illustrated in Figure C.3. Again, the interface numbering convention is to apply the 0-to-7 bit labels to the least- through most-significant bits within each received data byte. The correspondence of RXC<3:0> signals to associated data bytes is specified at the bottom of Figure C.3.

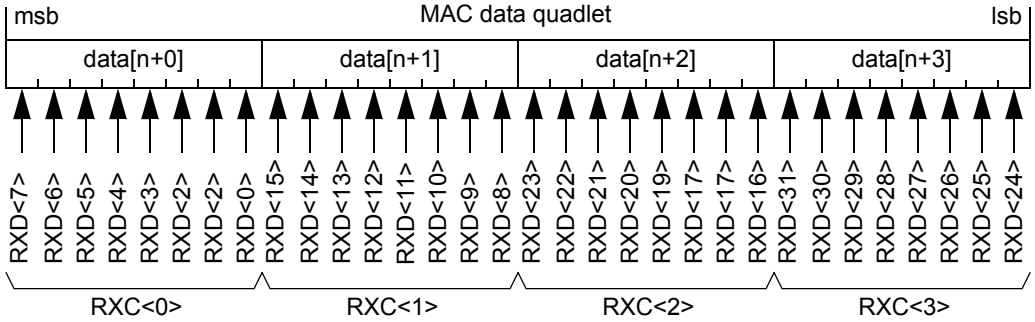


Figure C.3—XGERS receive signal mapping

Annex D.

(normative)

SONET PHY

D.1 ...

D.2 ...

D.2.1 ...

D.2.1.1 ...

D.2.1.2 ...

D.2.1.3 ...

D.2.1.4 ...

D.2.1.5 ...

Figure D.4—Gibabit ethernet reconciliation sublayer (GERS)

TDAT<7:0> are data-byte bits: the 0'th through 7'th bit are the first through last transmitted data bits.
TFCLK is asserted when a data byte is transmitted; this bits stays asserted during each frame transmission.
TSOP is TBD.
TEOP is TBD.

RDAT<7:0> are data-byte bits: the 0'th through 7'th bit are the first through last received data bits.
RFCLK is asserted when a data byte is transmitted; this bits stays asserted during each frame reception.
RFSOP is TBD.
TFEOP is TBD.

Reconcile following subclause signal names with figure names.

Annex E.

(informative)

CRC calculations

Annex F.

(normative)

802.1D and 802.1Q Bridging conformance

Editors' Notes (DVJ): *To be removed prior to final publication.*

The following should be updated in the terms and definitions:

bidirectional2 flooding -- a frame forwarding transfer involving sending two flooding frames, one on each ringlet, where the each frame is directed to distinct adjacent stations.

bidirectional1 flooding -- a frame forwarding transfer involving sending two flooding frames, one on each ringlet, where the each frames is directed to the same station.

unidirectional2 flooding -- a frame forwarding transfer involving sending a flooding frame in the down-stream direction, and that frame is directed to the station's upstream neighbor.

unidirectional1 flooding -- a frame forwarding transfer involving sending a flooding frame in the down-stream direction, and that frame is directed to its sending station.

remote unicast -- A transfer directed to a specific RPR station with the intent of forwarding a stripped version of that frame to a distinct remote node.

local unicast -- A transfer directed to a specific RPR end station.

flooded -- A multicast, broadcast, or unlearned remote-unicast frame sent from one station to all others.

Editors' Notes (DVJ): *To be removed prior to final publication.*

This writeup represents common text and two proposals for bridge-compatible formats, proposal P&Q, based on Vancouver meeting comments:

For both proposals:
1) Type field. The type field is moved to the payload, as suggested in unrelated comment.
2) Header coverage. A 32-bit HEC eliminates the 16-bit CRC loss-of-error-coverage concern.
3) Alignment. The frame header is now 32-bit aligned.

Other properties are different for proposal P & Q:
Proposal P:
1) Smaller. For remote transfers, the header is more compact.
2) Uniform. Local and remote formats are the same.
3) Reliant. The protocols rely on support of RPR-local DSID&SSID allocation during discovery.

Proposal Q:
1) Larger. For remote transfers, an additional sourceMacAddress field is prepended to the payload.
2) Strange. Local and remote formats are different, due to dependent placement of sourceMacAddress.
3) Ready. The protocols impose no requirements for DSID/SSID allocation during discovery.

Editors' Notes (DVJ): *To be removed prior to final publication.*

The term "run" might make more sense than "ringlet" when in the wrapping mode.

F.1 Flooding techniques

F.1.1 Multicast/broadcast forwarding

Editors' Notes (DVJ): *To be removed prior to final publication.*

The following discussion on multicast/broadcast forwarding probably belongs in the overview (with details in clause 6), but is being retained in this annex until new placement location is confirmed:

The most basic multicast/broadcast distribution techniques involves circulating a frame through all stations on the ring. The forwarding techniques for multicast/broadcast transfers are the same as those described for flooded frames, illustrated in Figure F.9 through Figure F.16.

NOTE—Stations are not expected to optimize the efficiency of multicast forwarding. To reduce complexities, they are expected either support unidirectional1 multicasts or to forward multicasts and flooded frames in the same fashion.

F.1.2 Flooding bridge transfers

Editors' Notes (DVJ): *To be removed prior to final publication.*

The following flooding text probably belongs in the overview, since its mostly informative.

Transparent bridging requires a form of one-to-others distribution called flooding. Flooding protocols (flooding, multicast, and broadcast) require the inclusion of additional information, beyond that included within the client-visible Ethernet frame. That information includes local destination station identifier (DSID) and local source station identifier (SSID) along with other maintenance/control fields. These addresses assist in scoping the range of the flooding distribution and suppressing undesirable duplicates that might otherwise be generated.

Bridges use the DSID/SSID addresses to flood (a flood is a type of broadcast) remote frames for delivery to all all bridge clients, as illustrated in the left side of Figure F.5. Flooding involves transmissions between a single source station and all other stations.

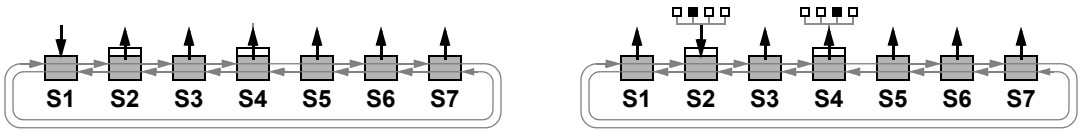


Figure F.5—Basic bridge flooding

With flooding, a frame is placed on the ring by the source, copied by intermediate stations, and stripped at the destination (depending on the method, the frame may also be copied in the destination). Flooded frames may pass through an additional multicast filter, which selectively rejects uninteresting multicasts, but this action is decoupled and independent from the flood protocols of the RPR ringlet.

Basic-bridging stations maintain simplicity by always flooding, as illustrated in Figure F.5. Although no spatial reuse is possible, this avoids overheads associated with maintaining and utilizing RPR forwarding tables.

F.1.3 Unicast considerations

Editors' Notes (DVJ): *To be removed prior to final publication.*

The following informative belongs in the overview. While enhanced bridging is not part of this specification, background information is useful to ensure that code space is sufficient for enhanced bridging extensions.

Local stations improve efficiencies by directing local-unicast traffic to the affected station, rather than flooding this traffic to all others, as illustrated in the left side of Figure F.6. The determination of whether to use flooded or unicast frames is based on a comparison of the frame's destinationMacAddress with the RPR topology database: a unicast is used if a local station matches the same destinationMacAddress; a flood is used otherwise.

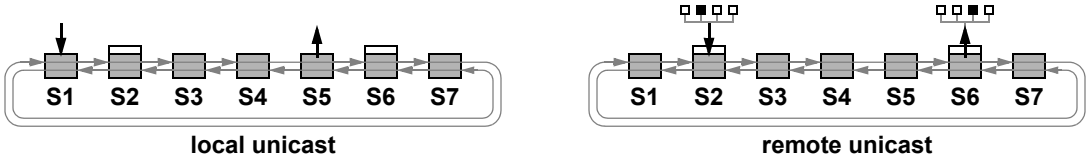


Figure F.6—Enhanced bridge unicasts

Enhanced-bridging stations improve efficiencies by maintaining and utilizing RPR forwarding tables, so that remote frames can also be unicast, as illustrated in the right side of Figure F.6. The learn improves link utilization, due to the frame's unicast (not flooded) and shortest-path nature.

Ordering constraints mandate that flooded and related remote-unicast transfers flow over the same path. The term *related* refers to fames with an identical set of *{sourceMacAddress/destinationMacAddress,VLAN}* identifiers. Flowing over the same path is necessary to maintain ordering, without invoking an inefficient flush between floods and related remote-unicast transfers.

For unidirectional flooding, the potential performance impact of this ordering constraint can be severe, in that the worst case path-length nearly doubles over that associated with bidirectional flooding. To avoid that potential performance impact, enhanced bridges are expected to support bidirectional flooding.

F.1.4 Flooding flow notation

A variety of remote-transfer flows are illustrated in following subclauses, as illustrated in Figure F.7. Downward and upward arrows identify client-to-MAC and MAC-to-client transfers respectively. Downward end-of-flow curves identify locations where frames are stripped. The *x* marker at the end of an error identifies locations where frames are discarded, due to detected inconsistency errors.

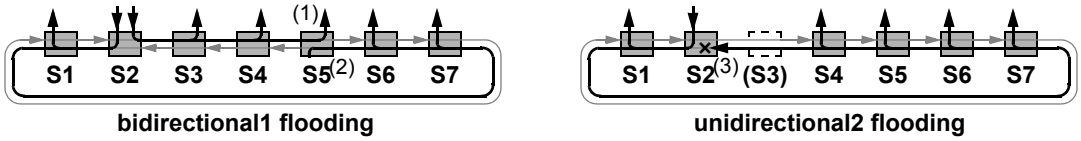


Figure F.7—Flooded receive operations

The frame header contains sufficient information to invoke (1) FLOOD_COPY or (2) FLOOD_TOSS operations, as illustrated on the top and bottom ringlets within stationE on the left side of Figure F.7. A frame deletion operation (3) may also occur, as illustrated within stationB on the right side of Figure F.7. The deletion operations is invoked by consistency-check errors.

F.2 Flooding alternatives

Several flooding alternatives are provided, as illustrated in Table F.1. Flooding entries within this table are ordered by complexity: the first-through-last entries are the most-through-least complex and most-through-least efficient (from a link-bandwidth utilization perspective). Although stations are expected to use the second (bidirectional1) alternative, they may use any of the four listed alternatives. Within this table, the *Reference* column provides a cross-reference to the applicable descriptive subclause.

Table F.1—Flooding alternatives

Name	DSID0	DSID1	Reference	Description
bidirectional2	midPoint0	midPoint1	F.2.2	Two floods, to distinct midpoint stations
bidirectional1	midPoint	midPoint	F.2.3	Two floods, but to the same midpoint station
unidirectional2	upstream	-na-	F.2.4	One downstream flood to the upstream station
unidirectional1	self	-na-	F.2.5	One flood to the source (*DSID=SSID*)

Lower efficiency is associated with flooding in both of two possible directions, rather than only one, since ringlet load balancing is a natural side-effect of shortest-path routing. Bidirectional2 flooding is more efficient; bidirectional1 flooding reduces the number of mid-point station identifiers from two-to-one at the cost of sending half of the transfers over an additional hop.

Lower efficiency is associated with flooding in only one of the two possible directions, rather than both. Simple load-balancing techniques could involve hashing the *sourceMacAddress*, *VLAN*, and *priority* fields to consistently select between westside and eastside paths. Unidirectional2 flooding is more efficient; unidirectional1 flooding eliminates the need to address one's upstream neighbor at the cost of sending transfers over an additional hop.

F.2.1 Flooding with steered protection

Flooding with steered protection is not influenced by the bidirectional2, bidirectional1, unidirectional2, or unidirectional1 nature of flood operations. Steered flooding involves concurrent transmissions with distinctive nonadjacent station S1 and station S7 failure-point destinations, as illustrated in Figure F.8. A FLOOD_COPY receive operation is specified for both of the steered transfers.

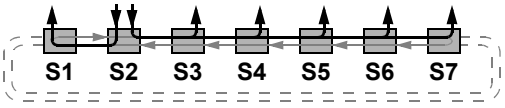


Figure F.8—Steered flooding

Flooding on rings is influenced by the bidirectional2, bidirectional1, unidirectional2, or unidirectional1 nature of flood operations, as discussed in the following subclauses. From the clients' perspective, flooding on unwrapped and wrapped rings has the same behavior, although the paths of frames changes due to the wrapping at failure points.

F.2.2 Bidirectional2 flooding

Bidirectional2 flooding of a ring involves concurrent transmissions on both ringlets, typically directed to a pair of mid-point station, as illustrated in the left and right sides of Figure F.9. A FLOOD_COPY receive operation is specified for westside as well as eastside transfers, causing the flooded frame to be passed to the client as each of its removal stations.

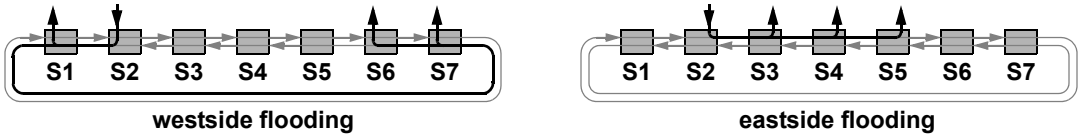


Figure F.9—Bidirectional2 flooding

Bidirectional2 flooding with wrap protection involves concurrent transmissions on both ringlets, typically directed to a pair of mid-point station, as illustrated in the left and right sides of Figure F.10. Again, a FLOOD_COPY receive operation is specified for westside as well as eastside transfers, causing the flooded frame to be passed to the client as each of its removal stations..

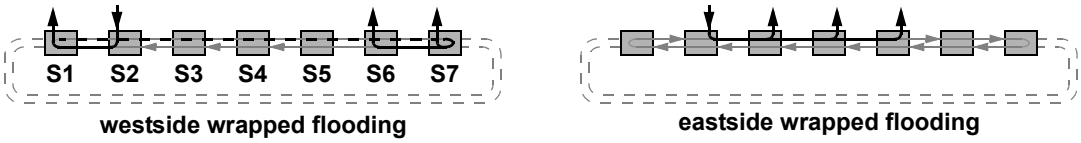


Figure F.10—Bidirectional2 flooding with wrapped protection

F.2.3 Bidirectional1 flooding

Bidirectional1 flooding of a ring involves concurrent transmissions on both ringlets, typically directed to a single mid-point station, as illustrated in the left and right sides of Figure F.11. A FLOOD_COPY receive operation is specified for one of the bidirectional1 transfers; a FLOOD_TOSS operation is specified for the other.

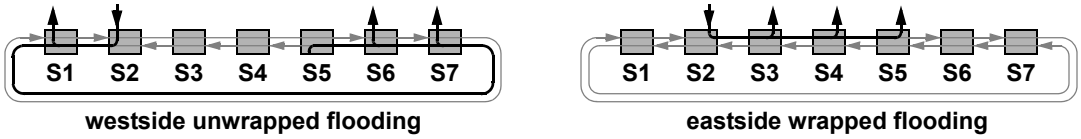


Figure F.11—Bidirectional1 flooding

Bidirectional1 flooding with wrapped protection involves concurrent transmissions to the common midpoint station, as illustrated in the left and right sides of Figure F.12. A FLOOD_COPY receive operation is specified for one of the bidirectional1 transfers; a FLOOD_TOSS operation is specified for the other.

Figure F.12—Bidirectional1 flooding with wrapped protection

F.2.4 Unidirectional2 flooding

Unidirectional2 flooding involves either a westside or eastside transmission directed to the source's upstream station, as illustrated in the left and right sides of Figure F.13. A FLOOD_COPY receive operation is specified, regardless of which direction is selected.

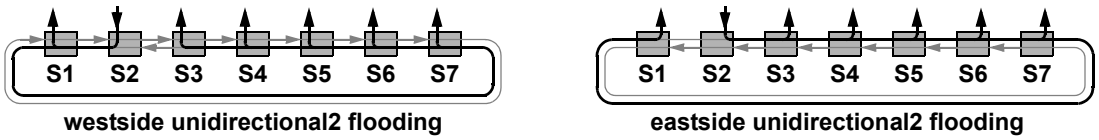


Figure F.13—Unidirectional2 flooding

Unidirectional2 flooding with wrapped protection involves a single transmission directed in either the westside or eastside directions, as illustrated in the left and right sides of Figure F.14. A FLOOD_COPY receive operation is specified, regardless of which direction is selected.

Figure F.14—Unidirectional2 flooding with wrapped protection

F.2.5 Unidirectional1 flooding

Unidirectional1 flooding involves either a westside or eastside transmission directed to the source station, as in the left and right side of Figure F.15 respectively. A FLOOD_TOSS receive operation is specified, regardless of which direction is selected.

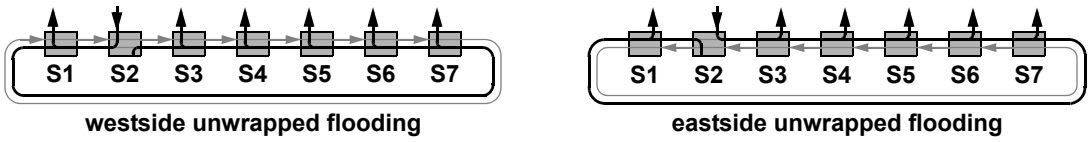


Figure F.15—Unidirectional1 flooding

Unidirectional1 flooding with wrapped protection involves either a westside or eastside transmission directed to the source station, as in the left and right side of Figure F.16 respectively. The wrapped flooding operation relies on the wrap capability at the endpoints. A FLOOD_TOSS receive operation is specified, regardless of which direction is selected.

Figure F.16—Unidirectional1 flooding with wrapped protection

F.2.6 Flood copy rules

Flooding involves selectively copying non-deleted primary-run frames to the client, as follows:

a) At intermediate stations, FLOOD_COPY&FLOOD_TOSS are copied to the client and forwarded to the downstream station.
b) At the destination station, FLOOD_COPY is copied to the client and stripped.
c) At the destination station, FLOOD_TOSS is not copied to the client, but is stripped.

F.2.7 Flooding constraints

Based on its flooding constraints, clause 6 of IEEE Std 802.1D-1998 architecture is summarized as follows:

a) Loss: A frame may be lost due to many reasons including topology change (subclause 6.3.2).
b) Misordering. Frame misordering is not allowed under any conditions (subclause 6.3.3).
c) Duplication. Frame duplication is not allowed under any conditions (subclause 6.3.4).

The misordering constraint can be met by having the client force a flush of in-progress transimissions before topology changes are allowed to affect the forwarding parameters (see F.4.2). In normal operation, the duplication constraint is met by flooding in nonoverlapping directions, as described in F.2.3. In abnormal operation, such as during topology changes the duplication constraint mandate consistency checks based on destination-station identifier (*DSID*) and source-station identifiers (*SSID*) values, as described in F.2.3.

F.3 Duplicate scenarios

F.3.1 Duplicate scenarios: Unidirectional source bypass

Unidirectional flooding is susceptable to a source-station-pair loss during flooding, as illustrated in Figure F.17. In this example, source-station *S2* and its upstream neighbor *S3* are both bypassed while the *S2*-sourced frame is circulating. Correct source-bypass processing involves discarding the frame when its recirculates beyond its virtual source, as illustrated by the *x* marks within these Figure F.17.

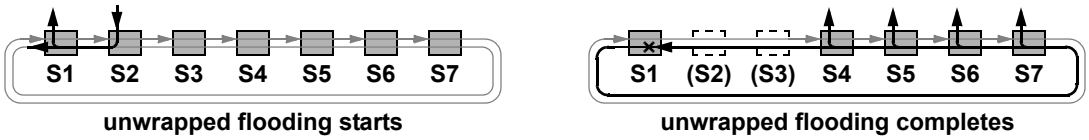


Figure F.17—Duplicate scenarios: Unidirectional source bypass

Cause: The source (that was responsible for packet deletion) disappears before its frame returns.
Problem: The packet passing through stations *S3*&*S2* may be falsely accepted by station *S1* (and others).
Solution: Discard packets if *(256-timeToLive)!=Hops[SSID]*.

F.3.2 Duplicate scenarios: Unidirectional wrapped source bypass

Unidirectional wrapped flooding is also susceptable to a source-station loss during flooding, as illustrated in Figure F.18. In this example, source-station *S2* and its upstream neighbor *S3* are both bypassed while the *S2*-sourced frame is circulating on the rightside of station *S3*. Correct source-bypass processing involves discarding others' transfers when recirculate beyond the source, as illustrated by the *x* marks within these Figure F.17.

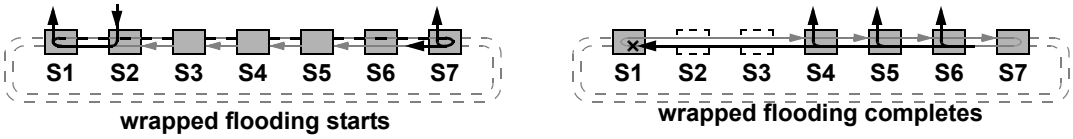


Figure F.18—Duplicate scenarios: Undirectional wrappes source bypass

Cause: The source (that was responsible for packet deletion) disappears before its frame returns.
Problem: The packet passing through station *S2* may be falsely accepted by station *S1* (and others).
Solution: Discard wrapped packets if *Hops[DSID]>0*.

F.3.3 Duplicate scenarios: Unidirectional destination bypass

Bidirectional flooding is susceptable to a destination-station-pair loss during flooding, as illustrated in Figure F.17. In this example, destination stations *S5*&*S6* are bypassed while the *S2*-sourced frame is circulating. Correct destination-bypass processing involves discarding the frame when its circulates beyond its virtual destination, as illustrated by the *x* marks within these Figure F.19.

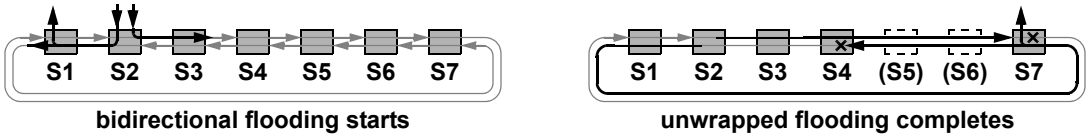


Figure F.19—Duplicate scenarios: Unidirectional destination bypass

Cause: The destination (that was responsible for packet deletion) disappears before its frame arrives.
Problem: The packet passing through stations *S5*&*S6* may be falsely duplicated at station *S4*, *S7*, and others.
Solution: Discard packets if *Hops[DSID]<Hops[SSID]*, where *Hops* is a database array.

F.3.4 Duplicate scenarios: Bidirectional destination removals

Bidirectional wrapped flooding is susceptable to a destination-station-pair loss during flooding, as illustrated in Figure F.20 In this example, destination stations *S5*&*S6* are removed while the *S2*-sourced frame is circulating. Correct destination-bypass processing involves discarding the frame when its circulates beyond its virtual destination, as illustrated by the *x* marks within these Figure F.20.

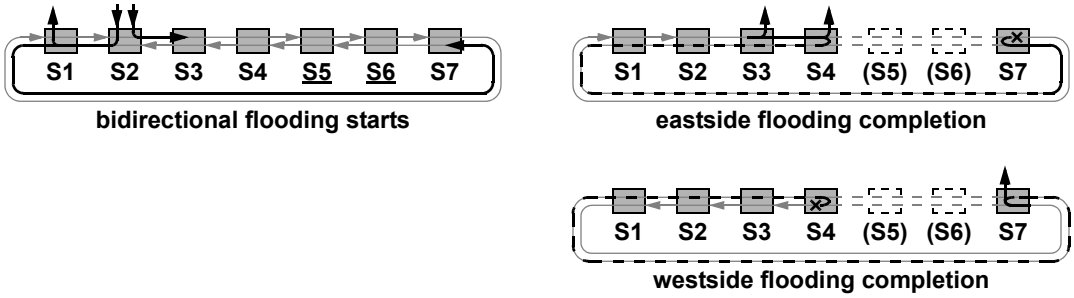


Figure F.20—Duplicate scenarios: Bidirectional destination removals

Cause: The destination (that was responsible for packet deletion) disappears before its frame arrives.
Problem: The packet wrapped before stations *S5*&*S6* may be falsely duplicated at station *S4*, *S7*, and others.
Solution: Discard wrapped frames if *Hops[DSID]>0*.

F.3.5 Duplicate scenarios: source&destination removals

Unidirectional flooding could be disrupted when half of the stationsn (including the source and destination stations) are removed, as illustrated in Figure F.21. In this example, source station *S2* along with stations *S1*, *S7*, and *S8* are removed while the *S2*-sourced frame is circulating. Correct processing involves discarding returning frames when their source is missed.

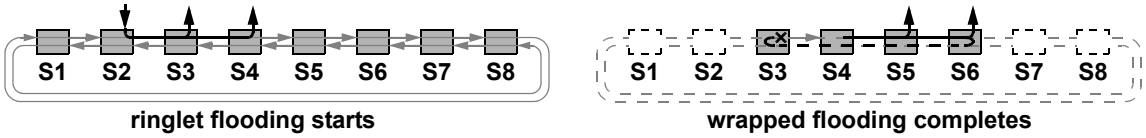


Figure F.21—Duplicate scenarios: source&destination removals

Cause: The source (that was responsible for packet deletion) disappears before its frame recirculates.
Problem: The packet may be falsely duplicated when recirculated to station *S3* and others.
Solution: Discard wrapped frames unless the source&destination (in that order) are present on the return.

F.4 Flushing requirements

F.4.1 Flushing residual frames

> ***Editors' Notes (DVJ):*** *To be removed prior to final publication.*
>
> A more extensive description of topology sequence numbers is needed.

Flushing frames involves incrementing the source's *tsn* (topology sequence number) field. This has the effect of causing stale flooded frames to be discarded, since the destination and frame *tsn* will differ.

F.4.2 Flushing during protection

F.4.2.1 Wrapped protection flushing

A link failure effects the routing (1) of packets that would normally pass over the affected link. If protection wrapping is employed, some failed-link traffic (2) is lost but no flushing is required, as illustrated in Figure F.22. Wrapped transmissions (3) continue in a transparent ordered fashion. During protected-link restoration, frames on the wrap-return path (4) are lost but (once again) no flushing is required.

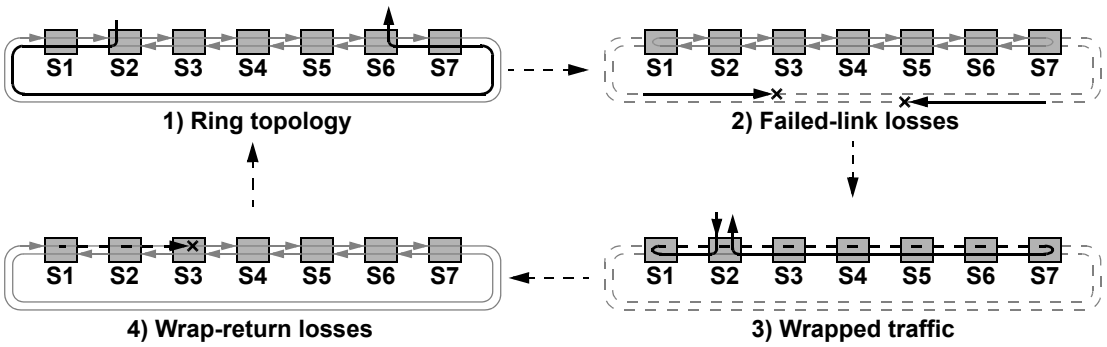


Figure F.22—Wrapped protection (no flushing required)

> ***Editors' Notes (DVJ):*** *To be removed prior to final publication.*
>
> Wrapping would be unnecessary is failure and restoration events were spaced far apart.
>
> There appears, however, to be a timing hazard associated with a quick restoration-and-failure sequence, since the return-run traffic from before the restoration will be unordered with respect to the after-failure traffic. More analysis of this timing hazard and potential solutions is needed.

F.4.2.2 Steered protection flushing

Within some applications, a more efficient protection-steering protocols may be desired. To maintain packet ordering when switching between normal and steering modes, outstanding traffic is flushed before steering is invoked. After the flush all packets are known to have be delivered, so that switching between normal and protection-steering is therefore safe.

A link failure effects the routing (1) of packets that would normally pass over the affected link. If protection steering is employed, some traffic will continue to be lost (2a) until intermediate stations become aware of the ring failure, as illustrated in Figure F.23. A flush (2b) is then required before stations can begin steered-transmissions on both ringlets, since pre-protection and post-protection traffic travels over distinct paths.

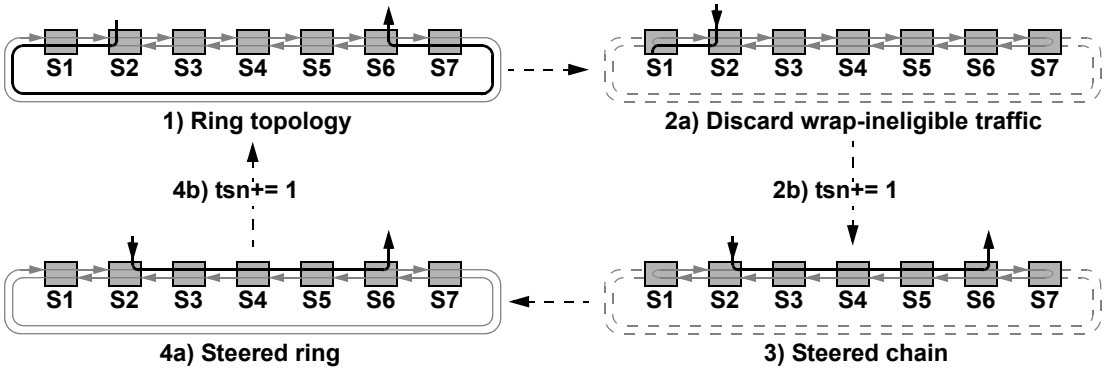


Figure F.23—Steered protection flushing

When the ring (4a) is restored, a flush (4b) is required before the steered ring transfers can be redirected over shortest-distance paths.

F.4.2.3 Wrap-then-steer protection flushing

Wrapping minimizes packet loss during the protection event, but consumes excessive link bandwidth until steering can be enabled. Immediate steering stalls protected transmissions until after a unscheduled flush operation can be performed. To avoid this unscheduled stall, a combination of immediate wrapping followed by scheduled steering is sometimes applicable.

A link failure effects the routing (1) of packets that would normally pass over the affected link. If protection wrapping is initially employed, a (2b) flush is required before the wrapped traffic (3) can be steered, as illustrated in Figure F.24.

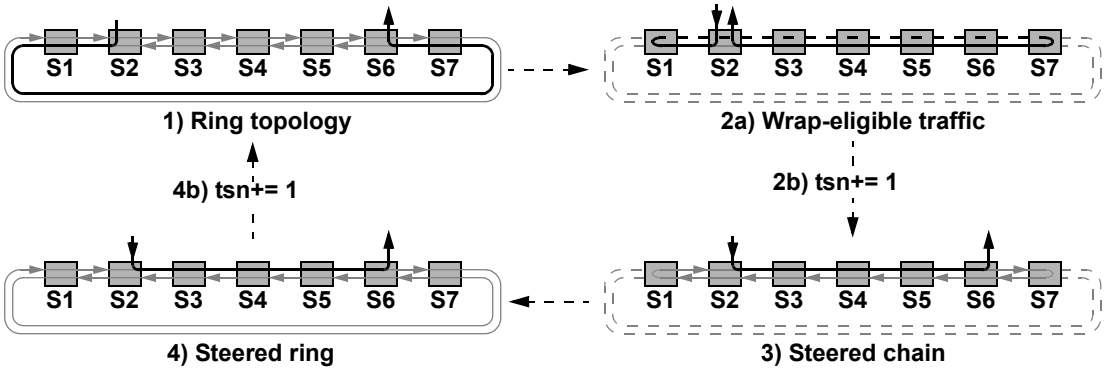


Figure F.24—Steered protection flushing

When the ring (4a) is restored, a flush (4b) is required before the protection-steered transfers can be redirected over shortest-distance paths.

F.4.3 Topology-change flushing

F.4.3.1 Detaching and attaching stations

On a normal ring (1), one or more of the attached stations (2a) can be detached, so that fewer stations appear connected to the ring, as illustrated in Figure F.25. A flushing step is required (2b) before revising shortest-path forwarding tables for the new topology (3).

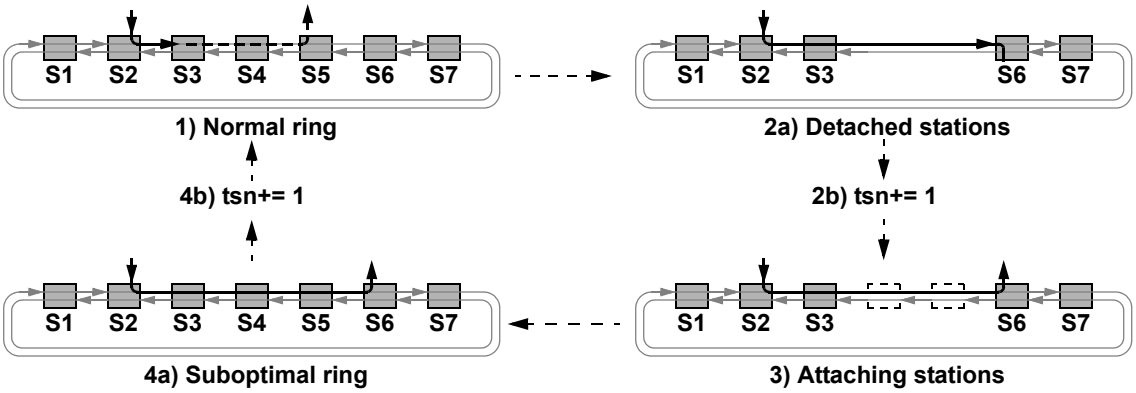


Figure F.25—Detaching and attaching stations

If intermediate stations (4a) are then attached, the optimality of ringlet-selection protocols is compromized. A flushing step is required (4b) before revising shortest-path forwarding tables for the new topology (1).

F.4.3.2 Split and joined chains

A second failure (1) in a ring (on a single failure in a chain) can form two distinct rings, as illustrated in Figure F.26. After separation, steered frames sent towards the severed chain (2a) are discarded at the wrap point. A flush (2b) is necessary to rid the interconnect of the disjoint station S5-to-S7 traffic, to prevent missordering if the station S1-to-S7 list is nearly simultaneously restored. Updates of the forwarding tables limit the steered transmissions to existing stations (3).

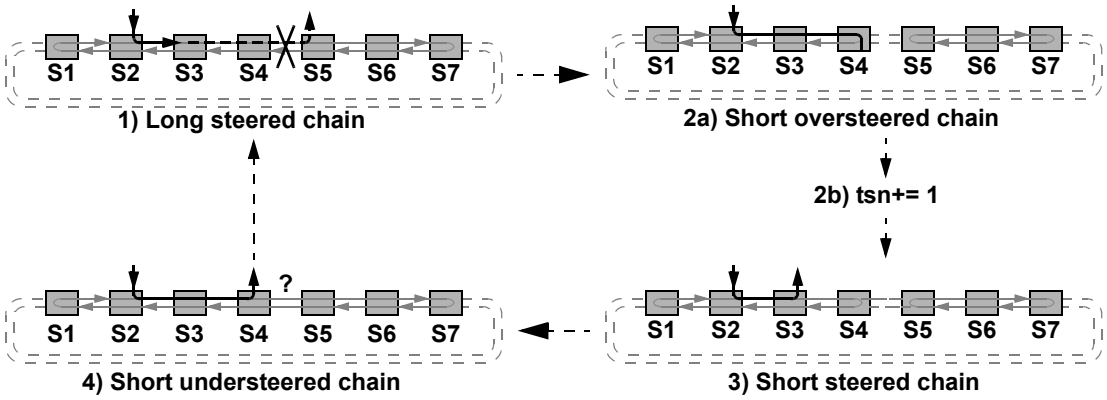


Figure F.26—Split and joined chains

The addition (4) of an additional chain makes all stations once-again visible, as illustrated in Figure F.26. Updates of the forwarding tables extend the steered transmissions to support reattached stations, fully restoring (1) the previously long-steered-chain topology. No flushing is necessary to support extended chains, since the curent forwarding table is a strict subset of the extended-topology forwarding table.

F.4.4 Flush management

A ringlet flush involves three sequential steps, listed below. To reduce MAC complexity and increase client flexibility, these operations are performed at the client, not the MAC.

a) Stopping client transmissions
b) Sending a flush frame to one's self
c) Resuming client transmissions.

The client has the option to consider whether mis-ordering and duplication can be optionally allowed (like 802.1w), when flows are explicitly known to explicity such reorderings. With such bimodal strategies, the client can support ordered-flow requirements (which suffer an additional flush-circulation delay) without delaying other unordered flows.

NOTE—Flushing after flooding would theoretically allow the use of shortest-path remote-unicast fowarding, after unidirectional1 flooding, but such an approach suffers from additional latency, delay jitter, and serialization complexities.

F.5 Frame aging

F.5.1 Basic *timeToLive* aging

Editors' Notes (DVJ): *To be removed prior to final publication.*

The following discussion on timeToLive aging better belongs in the overview (with details in clause 6), but is being retained in this annex until new placement location is confirmed.

The RPR protocols include the use of a *timeToLive* field within the frame header. This *timeToLive* field is set to 255 when the frame is first transmitted and is decremented when passing through each station, as illustrated in Figure F.27. This *timeToLive* field serves two purposes:

a) Cleanup. Frames are discarded when their *timeToLive* field exceeds the ring circumference.
b) The *timeToLive* field identifies the distance between the source and intermediate/final stations. This information has multiple uses:
   1) For topology discovery, *timeToLive* generates the index for a topology database write (see xx).
   2) For duplicate-deletion checks, *timeToLive* generates an topology database index (see yy).
   3) For fairness frames, *timeToLive* identifies the choke-point location (see xx).

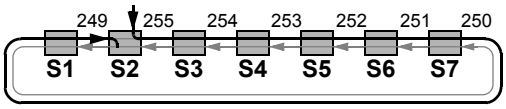


Figure F.27—Unwrapped *timeToLive* aging

F.5.2 Wrapped aging

Editors' Notes (DVJ): *To be removed prior to final publication.*
The following discussion on timeToLive aging better belongs in the overview (with details in clause 6).

For simplicity and uniformity, the *timeToLive* value is decremented when passing through each station. Additional actions are (a) associated with the wrap points and (b) source/destination return paths, as illustrated in Figure F.28.

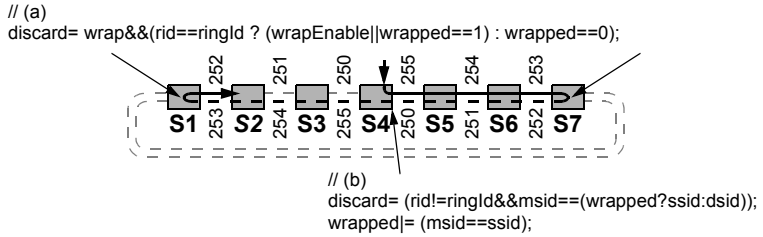


Figure F.28—Wrapped *timeToLive* updates

F.5.3 Intermediate duplicate-deletion rules

The duplicate-deletion actions are taken when the frame and attachment *ringID* values are the same, if any of the following deletion conditions occur.

a) Identifier deletion. The following two conditions are both satisfied:
   1) *frame.dsid*!=*frame.ssid* (frame was directed to another).
   2) *frame.ssid*==*database.msid* (the frame returned to the source).
b) Duplicate deletion (see F.6.4).
c) If not deleted, the copying of flooded frames depends on their type:
   1) A FLOOD_COPY flooded frame is copied to the client.
   2) A FLOOD_TOSS flooded frame is copied to the client on all but *dsid*-matching stations.
d) The flooded frames with a matching *dsid* address are then stripped.

NOTE—The scope deletion rules are less stringent than flooded deletion rules, as they are only intended to limit frame lifetimes to ensure correctness of a following flush operation. The scope rules are not intended to avoid duplicate unicast observations, since matching unicast observations are stripped and therefore cannot be duplicated. The more stringent flooded deletion rules are intended to ensure that no station will observe the same flooded frame more than once. Nonduplicated flooded frames will sometimes be dropped to fulfill this no-duplicate-frames objective.

F.6 Option P

F.6.1 Option-P: Uniform format conversions

Remotely-sourced transfers involve prepending of leading *timeToLive*, *flags*, *DSID*, and *SSID* information, along with 48-bit *destinationMacAddress* and *sourceMacAddress* components to ensure reliable RPR-local delivery, as illustrated in Figure F.29.

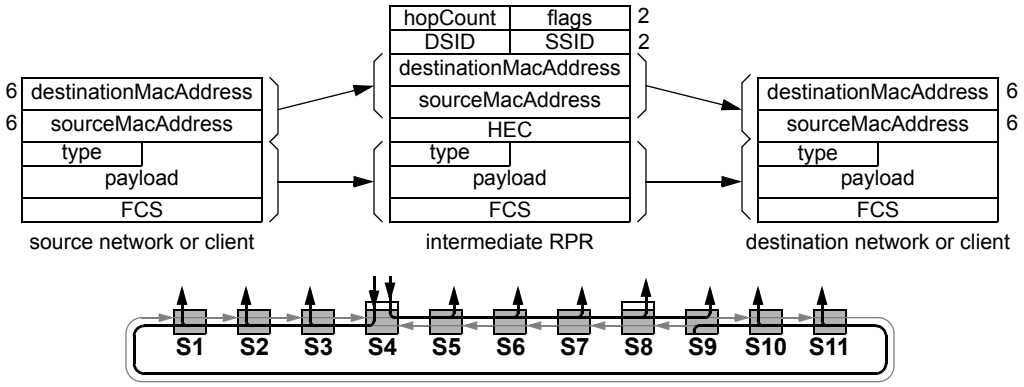


Figure F.29—Option-P: Frame forwarding

F.6.2 Option-P: Frame sourcing rules

Before a frame is transmitted, route-dependent information is placed within frames, as listed below.

a) *frame.timeToLive=255* (the default initial timeout value)
b) *frame.wrapped=0* (the frame has not yet been wrapped)
c) *frame.SSID=MSID* (the source station identifier)
d) Other information that depends on how the transfer method:
    1) Local or global unicast.
        i) *frame.type*=DIRECT_DATA
        ii) *frame.DSID=Map(destinationMacAddress);*
        iii) *frame.scope=Distance(SSID,DSID)* (hop counts between source and destination)
    2) Unidirectional1 flooding.
        i) *frame.type*=FLOOD_TOSS
        ii) *frame.DSID=Map(sourceMacAddress)*;
        iii) *frame.scope= leftSpan+rightSpan;*
    3) Bidirectional1 flooding (independent operations for ringlet-0 ringlet-1):
        i) *frame.type*=FLOOD_TOSS
        ii) *frame.DSID=Map(midMacAddress)*; (*midMacAddress* is an equal-distance station)
        iii) *frame.scope=Distance(SSID,DSID)* (hop counts between source and destination)

F.6.3 Option-P: Duplicate deletions

Duplication deletion involves two type of consistency checks, as illustrated in Figure F.30. One type of check (e) is performed on the unwrapped portion of a ring. A second type of check (f) is performed on an the wrapped portion of a wrapped ring.

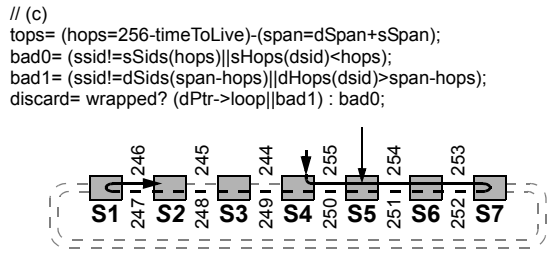


Figure F.30—Option-P: duplicate deletions

Duplicate deletions involves special frame processing at the wrap points, as follows:

```
// Process passing-through option-Q data frames
int
ProcessPFrame(DataBase *dPtr, Frame *fPtr)
{   uInt1 same, ssid, dsid, wrapped, loop, msid, hops, span, tops, bad0, bad1, dump;

    assert(dPtr!=NULL&&fPtr!=NULL);                          // Debug consistency check
    same= (fPtr->ri==dPtr->ri);                              // Establishing short names
    ssid= fPtr->ssid;  dsid= fPtr->dsid;                     //  simplifies the code but
    wrapped= fPtr->wrapped;                                  //  doesn't complicate the
    loop= dPtr->loop;  msid= dPtr->msid;                     //  actual hardware

    hops= (256-fPtr->timeToLive);                            // Actual source distance
    span= (dPtr->sSpan+dPtr->dSpan);                         // Sequential station count
    tops= hops-span;                                         // Wrap distance adjustment
    bad0= (ssid!=sSid(hops)||sHops(dsid)<hops);              // Initial run checks
    bad1= (ssid!=dSids(span-tops)||dHops(dsid)>span-tops);   // Wrapped run checks

    dump= wrap&&(same ? (wrapEnable||wrapped):!wrapped);     // Wrap & return checking
    dump|=  same&&(wrapped ? (loop||bad1):bad0);             // Primary run discards
    domp|= !same&&(msid==(fPtr->wrapped ? ssid:dsid));       // Secondary run discards
    fPtr->wrapped= (same==0&&ssid==msid);                    // Set wrapped on return
    fPtr->timeToLive-= 1;                                    // Decrement timeToLive
    return(dump);
}
```

F.6.4 Option-P: Data frame format

Within a complete frame, the header includes 6-bit *DSID,* 6-bit *SSID*, 48-bit *destinationMacAddress,* and 48-bit *sourceMacAddress* fields, as illustrated in Figure F.31.

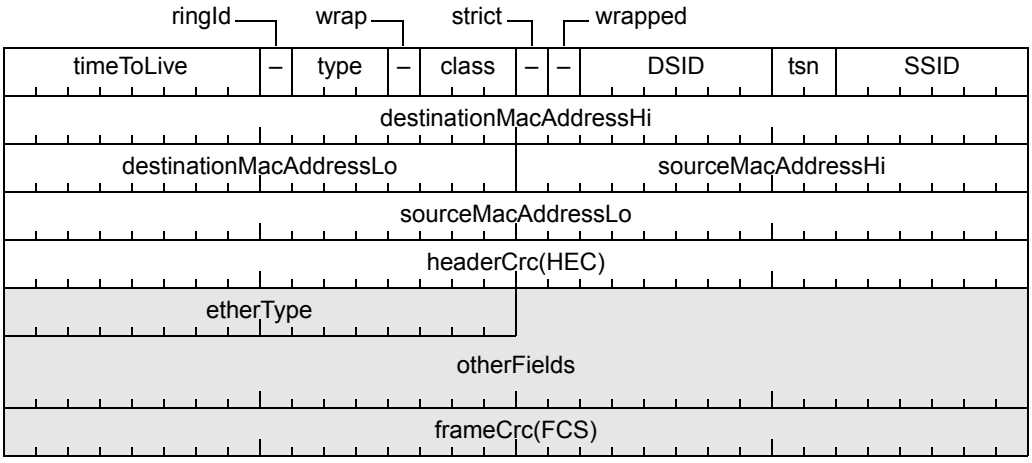


Figure F.31—Option P: Data frame format

The 8-bit ***timeToLive*** field is set to 255 when a frame is first transferred; the incoming value is decremented while the frame passes through other stations.

The ***ringID*** bit values of 0 and 1 indicate the frame was sourced on ring-0 and ring-1 respectively. The 3-bit ***type*** field specifies the frame format and forwarding features, as specified in F.6.5.

The ***wrap*** bit values of 0 and 1 identify wrap-ineligible and wrap-eligible frames respectively. The 3-bit ***class*** field values specify the class of RPR traffic, as specified in F.6.6.

The ***strict*** bit values of 0 and 1 enable and disable flood duplicate deletions respectively. The ***wrapped*** bit is initially zero and is one when a wrapped frame recirculates. The 6-bit ***DSID*** field identifies the ringlet-local destination-station.

The 2-bit ***tsn*** (topology sequence number) field values are used to enforce ordering constraints (see xx). The 6-bit ***SSID*** field identifies the ringlet-local source-station.

NOTE—The intent of the strict bit is to allow clients to selectively disable duplicate-deletion operations for applications known to be more sensitive to frame losses than frame duplications.

The concatenation of the 32-bit ***destinationMacAddressHi*** and 16-bit ***destinationMacAddressLo*** fields form the 48-bit MAC address of the destination station (this may or may not be located on RPR).

The concatenation of the 16-bit ***sourceMacAddressHi*** and 32-bit ***sourceMacAddressLo*** fields form the 48-bit MAC address of the remote source station (this may or may not be located on RPR).

F.6.5 Frame type format

The 3-bit ***type*** field specifies the frame format and forwarding features, as specified in Table F.2**.**

Table F.2—Frame *type* field values

Value	Name	Row	Description
0	DIRECT_FAIR1	F.2.1	Fairness control frame, type-1
1	DIRECT_FAIR2	F.2.2	Fairness control frame, type-2
2	DIRECT_CONT	F.2.3	Generic control frame
3	DIRECT_DATA	F.2.4	Directed client-data frame
4	FLOOD_TOSS_A	F.2.5	Basic flooding, strip-at-destination
5	FLOOD_COPY_A	F.2.6	Basic flooding, copy-strip at destination
6	FLOOD_TOSS_B	F.2.7	Enhanced flooding, strip-at-destination
7	FLOOD_COPY_B	F.2.8	Enhanced flooding, copy-strip at destination

Row F.2.1: A unicast fairness-type-1 control frame directed to the MAC control function.
Row F.2.2: A unicast fairness-type-2 control frame directed to the MAC control function.
Row F.2.3: A unicast control frame directed to the MAC.
Row F.2.4: A unicast data frame directed to the client.

Row F.2.5: A flooded baseline-sourced data frame that excludes the DSID location.
Row F.2.6: A flooded baseline-sourced data frame that includes the DSID location.
Row F.2.7: A flooded enhanced-sourced data frame that excludes the DSID location.
Row F.2.8: A flooded enhanced-sourced data frame that includes the DSID location.

NOTE—To ensure compatibility with future revisions of this standard, the basic-bridging receive behaviors when observing FLOOD_TOSS_B and FLOOD_COPY_B commands are defined (they are equivalent to the FLOOD_TOSS_A and FLOOD_COPY_A commands respectively). Future revisions of this standard are expected to define the conditions when these (currently reserved) codes may be generated.

F.6.6 Frame *class* format

The 3-bit ***class*** field value specifies the class of RPR traffic, as specified in Figure F.3.

Table F.3—*class* field values

Value	Name	Description
0-1	—	Reserved
2	CLASS_A0	SubclassA0 (baseline) traffic
3	CLASS_A1	SubclassA1 (STQ option generated) traffic
4	CLASS_B0	SubclassB0 (within profile) traffic
5	CLASS_B1	SubclassB1 (out-of profile) traffic
6	CLASS_C	ClassC weighted fairness traffic
7	—	Reserved

F.7 Option Q

F.7.1 Option-Q: Local formats

Local transfers involve prepending of leading *timeToLive*, *flags*, *scope*, and *tsn* information, to ensure reliable RPR-local delivery, as illustrated in Figure F.32.

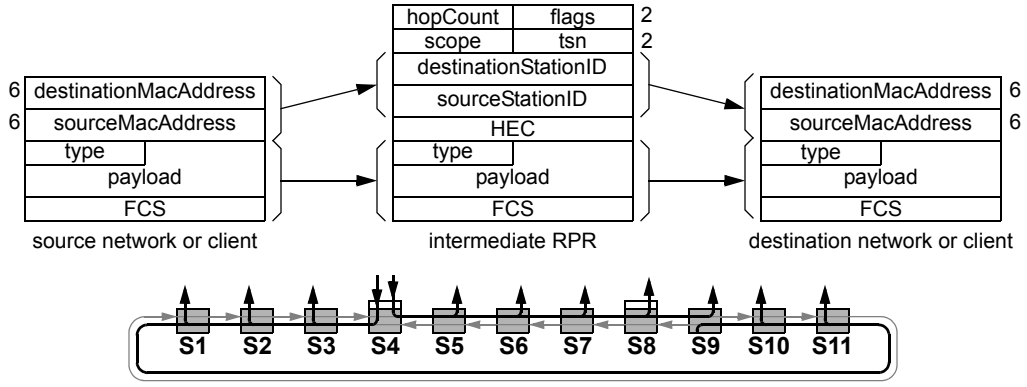


Figure F.32—Option-Q: Local frame forwarding

Remotely-sourced transfers involve prepending of leading *timeToLive*, *flags*, *scope*, and *tsn* information, along with 48-bit *destinationStationID* and *sourceStationID* components to ensure reliable RPR-local delivery, as illustrated in Figure F.33.

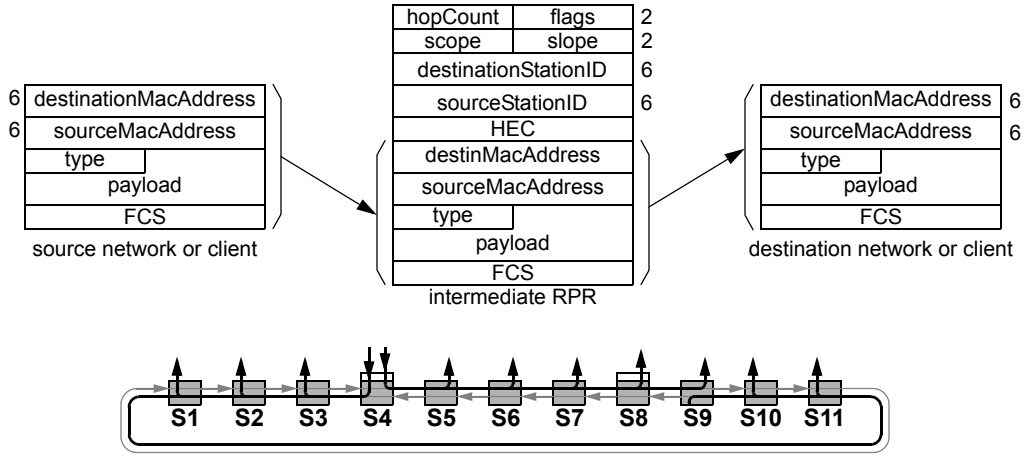


Figure F.33—Option-Q: Frame forwarding

F.7.2 Option-Q: Duplicate deletions

Duplication deletion involves two type of consistency checks, as illustrated in Figure F.30. One type of check (e) is performed on the unwrapped portion of a ring. A second type of check (f) is performed on an the wrapped portion of a wrapped ring.

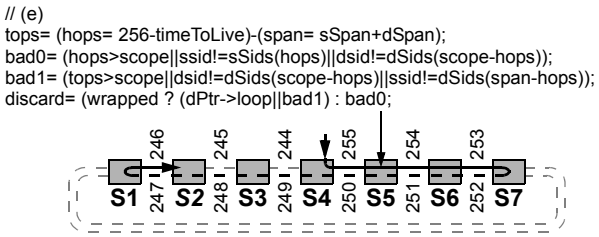


Figure F.34—Option-Q: Duplicate deletions

F.7.3 Option Q: Duplicate deletion code

Duplicate deletions involves special frame processing at the wrap points, as follows:

```
// Process discovery frames received from opposing ringlet
int
ProcessQFrame(DataBase *dPtr, Frame *fPtr)
{   uInt1 same, ssid, dsid, wrapped, loop, msid, hops, span, tops, bad0, bad1, dump;

    assert(dPtr!=NULL&&fPtr!=NULL);                          // Debug consistency check
    same= (fPtr->ri==dPtr->ri);                              // Establishing short names
    ssid= fPtr->ssid;  dsid= fPtr->dsid;                     //  simplifies the code but
    wrapped= fPtr->wrapped;                                  //  doesn't complicate the
    loop= dPtr->loop;  msid= dPtr->msid;                     //  actual hardware

    hops= (256-fPtr->timeToLive);                            // Actual source distance
    span= (dPtr->sSpan+dPtr->dSpan);                         // Sequential station count
    tops= hops-span;                                         // Wrap distance adjustment
    bad0= (hops>scope||ssid!=sSid(hops);                     // Initial run checks
    bad1= (tops>scope||dsid!=dSid(scope-tops));              // Wrapped run checks

    dump= wrap&&(same ? (wrapEnable||wrapped):!wrapped);     // Wrap & return checking
    dump|= !same&&(msid==(fPtr->wrapped ? ssid:dsid));       // Secondary run discards
    dump|=  same&&(wrapped ? (loop||bad1):bad0);             // Primary run discards
    fPtr->wrapped= (same==0&&ssid==msid);                    // Set wrapped on return
    fPtr->timeToLive-= 1;                                    // Decrement timeToLive
    return(dump);
}
```

F.7.4 Option-Q: Local-source format

Within a complete frame, the header includes 6-bit *scope*, 8-bit *tsn* (topology sequence number), 48-bit *destinationStationIdentifier,* and 48-bit *sourceStationIdentifier* fields, as illustrated in Figure F.36.

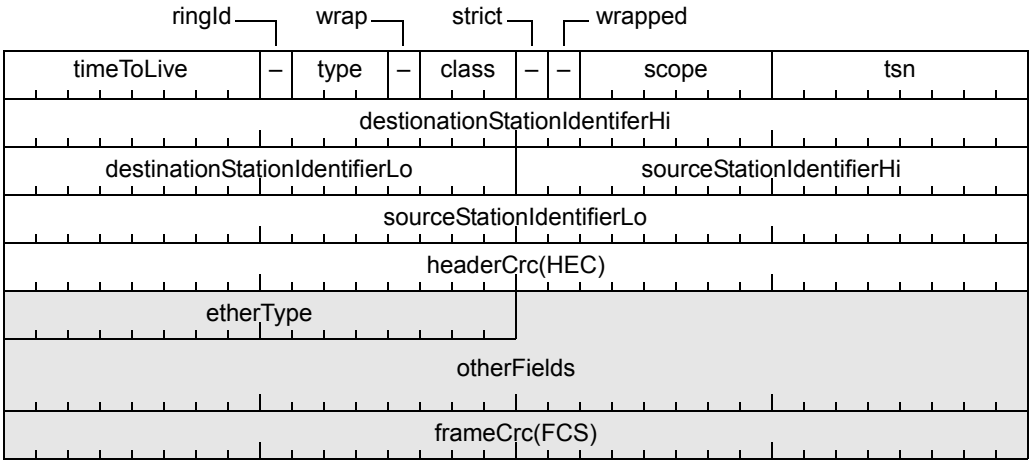


Figure F.35—Option-Q: Local-source format

The 8-bit ***timeToLive*** field, ***ringID*** bit, 3-bit ***type*** field, ***wrap*** bit, 3-bit ***class*** field, ***strict*** bit, ***wrapped*** bit, and 6-bit ***scope*** field are specified in F.6.4.

The 8-bit ***tsn*** (topology sequence number) field specifies a sequence number to support flushing operations.

The concatenation of the 32-bit ***destinationStationIdentifierHi*** and 16-bit ***destinationStationIdentifierLo*** fields form the 48-bit MAC address of the destination station (that may or may not be located on RPR). For this format, this value equals the *destinationMacAddress* identifier.

The concatenation of the 16-bit ***sourceStationIdentifierHi*** and 32-bit ***sourceStationIdentifierLo*** fields form the 48-bit MAC address of the local source station (this is an RPR station). For this format, this value equals the *sourceMacAddress* identifier.

F.7.5 Option-Q: Global frame format

Within a complete frame, the header includes 6-bit *DSID,* 6-bit *SSID*, 48-bit *destinationMacAddress,* and 48-bit *sourceMacAddress* fields, as illustrated in Figure F.36.

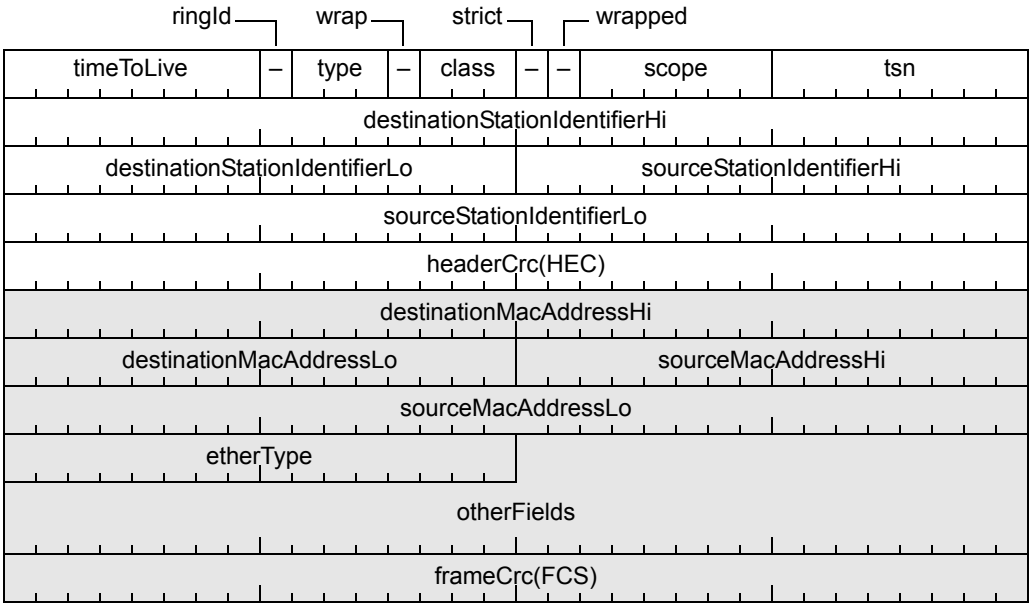


Figure F.36—Option-Q: Global frame format

The 8-bit ***timeToLive*** field, ***ringID*** bit, 3-bit ***type*** field, ***wrap*** bit, 3-bit ***class*** field, ***strict*** bit, ***wrapped*** bit, 6-bit ***scope*** field, and 8-bit ***tsn*** (topology sequence number) are specified in F.6.4.

The concatenation of the 32-bit ***destinationMacAddressHi*** and 16-bit ***destinationMacAddressLo*** fields form the 48-bit MAC address of the destination station (that may or may not be located on RPR).

The concatenation of the 16-bit ***sourceStationIdentifierHi*** and 32-bit ***sourceStationIdentifierLo*** fields form the 48-bit MAC address of the sourcing station (this is an RPR station).

The concatenation of the 32-bit ***sourceMacAddressHi*** and 16-bit ***sourceMacAddressLo*** fields form the 48-bit MAC address of the remote source station (this is not an RPR station).

F.8 Topology discovery

Editors' Notes (DVJ): *To be removed prior to final publication.*

The use of DSID and SSID address is based on the assumption that these can be easily derived during topology discovery. This subclause is intended to validate that assumption, by providing details on how DSID/SSID assignments could be performed.

F.8.1 Topology database parameters

The MAC has topology database components that assist in routing, as illustrated in Figure 0.1. The *sidInfo* array specifies station identifiers corresponding to each of the possible hop-count distance. The *hopInfo* array specifies hop-count distances and *sourceMacAddress* values associated with each station identifier.

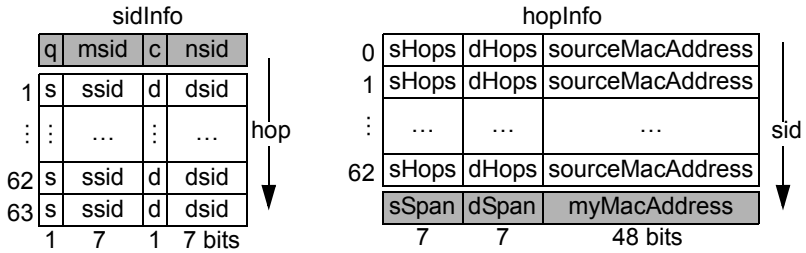


Figure 0.1—Topology database parameters

The gray information is not explicitly associated with either array, but may be stored within the otherwise unused first and last entries of the sidInfo and hopInfo arrays respectively. The ***q*** (queried) bit indicates this station shall continue rapid discovery transmissions. The ***msid*** (my station identifier) field is the normally ringlet identifier placed within this station's discovery frames. The ***c*** (chain) bit values of 0 and 1 correspond to discovered loop and chain topologies respectively. The ***nsid*** (next station identifier) field identifies the next available station identifier (*nsid* is used to resolve conflicts when more than one station has the same *msid* assignment).

The *sSpan* and *dSpan* fields specify the length of discovered in the source and downstream destination directions respectively. The myMacAddress field is the unique MAC address assigned to this station.

The ***s*** and ***d*** bits are set when any inconsistency is discovered and cleared when that entry is next updated. The station continues to transmit high-rate discovery messages until all *s* and *d* bits have been successfully cleared, a condition normally associated with stabilized topology database. The ***ssid*** and ***dsid*** fields associate station identifiers with the source-side and destination-side distances (their hop counts form the array index).

The ***sHops*** and ***dHops*** fields associate source-side and destination-side hop-count distances with the station identifier (the station identifier is the array index). The ***sourceMacAddress*** fields associate a 48-bit unique MAC address with each station identifier, so that duplicate assignments can be readily detected and corrected.

Since the database components are sometimes accessed at line-rate speeds, each attachment is expected to have a distinct copy of the topology database. This avoids mandating a high-speed data path between attachments, at a modest cost of supporting low-rate attachment-to-attachment discovery-frame communications.

F.8.2 Discovery frames

The 28-byte discovery frame transports only control flags within the payload, as illustrated in Figure F.37. The header provides most of the useful parameters, including the hop-count to the source (256-*timeToLive*), the source-station identifier *SSID*, and the MAC address of the source *sourceMacAddress*.

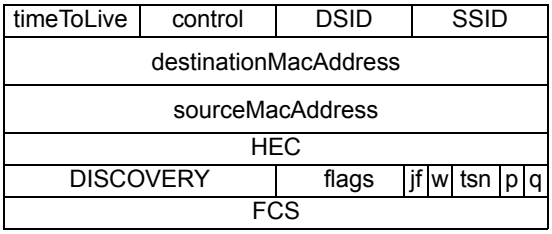


Figure F.37—Discovery frame components

Editors' Notes (DVJ): *To be removed prior to final publication.*

To support plug-and-play, the discovery frame should include *ri* (a hardwired ringlet identifier) and *rj* (and soft configured ringlet identifier). Simple refinements to the protocols would set all *ri*'s to the *ri* of the station with the largest MAC address; the *rj* values are provided for diagnostic purposes.

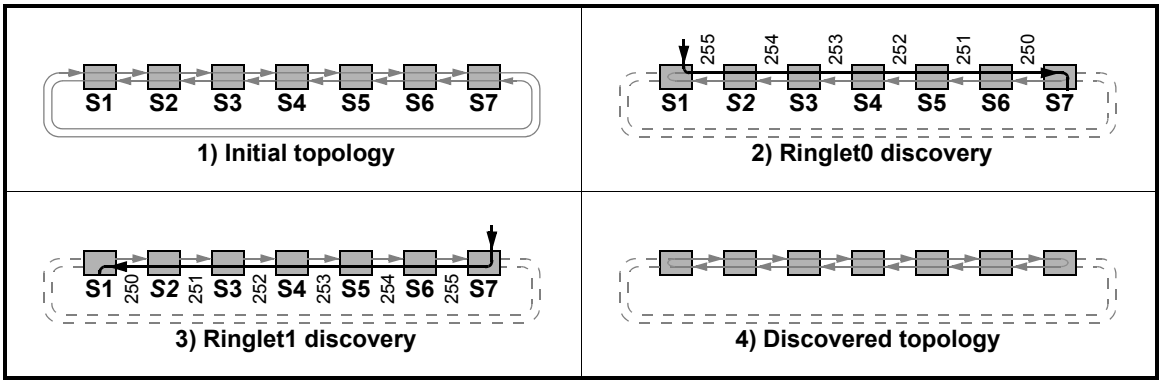
The discovery protocols provide header-resident identifiers, but basic capabilities are communicated within the payload. The ***jf*** (jumbo frame) bit values of 0 and 1 identify stations that support standard and jumbo frames respectively. The ***w*** (wrapping capable) bit values of 0 and 1 identify stations that are wrapping-capable and steering-only respectively.

The discovery protocols utilize the header parameters, but their actions are dependent on other payload-field components. The 2-bit ***tsn*** (topology sequence number) field is used to label flooded frames, so that the older no-longer-valid frames can be safely discarded. The ***p*** (protection) bit is one within frames sent from stations with a disabled opposing link. The ***q*** (query) bit is 1 when additional discovery transmissions are requested; otherwise *q* is 0.

F.8.3 Wrap discovery

During a protection event, each endpoints initiates a unidirectional transfers, as illustrated in Figure F.38. Although additional background topology messages are sent, these two transfers are typically sufficient to update the topology database.

Figure F.38—Wrapped ring discovery

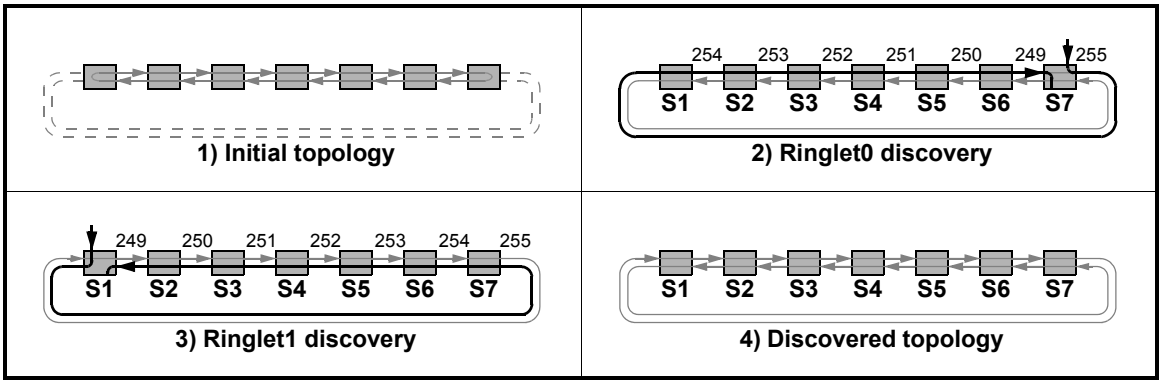


The hops value within the DSID properties array (see F.8.1) uses positive values to represent hop-count distances from upstream stations. Communication from the other attachment point, allows mid-span stations to learn the downstream portion of their new topology. Hop-count distances to downstream stations are represented as a negative hop-count distance, to simplify associated data-base updates.

F.8.4 Ring discovery

After a healing event, the endpoints initiate looped transfers, as illustrated in Figure F.38. These trigger other looped transfers, until each station has communicated with itself. Each station uses the receipt of its own looped message to confirm the loop nature of the interconnect.

Figure F.39—Unwrapped ring discovery



F.9 Database updates

F.9.1 DiscoverThis updates

When a discovery message is received on the attachment ringlet, the database is restructured based on the received information, as illustrated below.

```
// Process discovery frames received from attached ringlet
// The correct operation assumes:
// 1) Each station sends discovery frames, with q=1, while an s or d bit is set
// 2) All check if q==1; if so, they clear q=0 bit and transmit a fast discovery frame
// TBDs: a) Detect excessive-length loop or chain b) Dynamic ringID assignments
void
DiscoverThis(DataBase *dPtr, Frame *fPtr)
{   SidInfo *sPtr; uInt8 srcMac, myMac;
    uInt1 sid, hops, sHops, sSpan, dSpan, best, lost, sEnd, dEnd, ssid, i;

    GetArgs(dPtr,fPtr,&sid,&hops,&sSpan,&dSpan,&best,&lost,&srcMac,&myMac);
    if (hops>SMAX)
        return;
    ssid= dPtr->sidInfo[hops].ssid;
    sHops= dPtr->hopInfo[sid].sHops;
    if (myMac==srcMacMac) {
        dPtr->c= 0;                                                 // Self makes chain
        sEnd= dEnd= hops;                                           //  w/symmetric lists
    } else if (fPtr->protect) {
        dEnd= (dPtr->c==0) ? dEnd-hops:dSpan;                       // Sometimes 2 breaks
        dPtr->c= 1;                                                 // Loop is broken w/
        sEnd= hops;                                                 // protection hops
    } else {
        sEnd= sSpan;                                                // Span remains the
        dEnd= dSpan;                                                //  same by default
    }
    for (sPtr= &(dPtr->sidInfo[i=sEnd+1]); i<=sSpan; i+=1,sPtr+=1)
        sPtr->s= 0, sPtr->ssid= SMAX;                               // Src-side reduction
    for (sPtr= &(dPtr->sidInfo[i=dEnd+1]); i<=dSpan; i+=1,sPtr+=1)
        sPtr->d= 0, sPtr->dsid= SMAX;                               // Dst-side reduction

    dPtr->sSpan= sEnd;                                              // Set src-side span
    dPtr->dSpan= dEnd;                                              // Set dst-side span
    dPtr->sidInfo[hops].ssid= sid;                                  // Set src-side SID
    dPtr->hopInfo[sid].sHops= hops;                                 // Set src-side hops
    dPtr->hopInfo[sid].sourceMacAddress= best ? myMac:srcMac;       // Dup MAC overwrite
    if (lost)
        dPtr->nsid= ((dPtr->msid=dPtr->nsid)+1)%SMAX;               // Next available sid
    if (best!=0||lost!=0||sEnd!=sSpan||dEnd!=dSpan||sid!=ssid||hops!=sHops)
        for (i=1; i<SIDS; i+= 1)  dPtr->sidInfo[i].s= (i<=next);    // Change => queries
    dPtr->sidInfo[hops].s= 0;                                       // Cancel this query,
    dPtr->q|= fPtr->q;                                              //  but catch others
}
```

```
#define SIDS 64
#define SMAX (SIDS-1)
#define BASE 0
void
GetArgs(DataBase *dPtr, Frame *fPtr, uInt1 *sid, uInt1 *hops,
  uInt1 sSpan, uInt1 dSpan, uInt1 *best, uInt1 *lost, uInt8 *srcMac, uInt8 *myMac)
{   uInt8 srcMacAddress, myMacAddress; uInt1 same;

    assert(dPtr!=NULL && fPtr!=NULL);
    srcMacAddress= MERGE_SRC(fPtr->sourceMacAddressHi,fPtr->sourceMacAddressLo);
    myMacAddress= dPtr->myMacAddress;
    *sid= fPtr->ssid;
    *hops= 256-fPtr->timeToLive;
    *sSpan= dPtr->sSpan;
    *dSpan= dPtr->dSpan;
    same= (fPtr->ssid==dPtr->msid);
    *best= (conflict&&myMacAddress>srcMacAddress);
    *lost= (conflict&&myMacAddress<srcMacAddress);
    *srcMac= srcMacAddress;
    *myMac= myMacAddress;
}

// Advance nsid to the next invalid station-identifier entry
void
ScrubSids(DataBase *dPtr)
{   uInt1 nsid, sHops, dHops, sSpan, dSpan, ssid, dsid, srcGood, dstGood;

    nsid= dPtr->nsid;
    if (hops>SMAX)
        return;
    sSpan= dPtr->sSpan;
    dSpan= dPtr->sSpan;
    sHops= dPtr->hopInfo[nsid].sHops;
    dHops= dPtr->hopInfo[nsid].dHops;
    ssid= dPtr->sidInfo[sHops].ssid;
    dsid= dPtr->sidInfo[dHops].dsid;
    srcGood= (sHops<=sSpan&&ssid==nsid);
    dstGood= (dHops<=dSpan&&dsid==nsid);
    if (srcGood||dstGood)
        return;
    dPtr->nsid= (nsid+1)%SMAX;
}
```

F.9.2 DiscoverThat updates

When a discovery message is received on the opposing ringlet, the database is restructured based on the received information, as illustrated below.

```
// Process discovery frames received from opposing ringlet
void
DiscoverThat(DataBase *dPtr, Frame *fPtr)
{   SidInfo *sPtr; uInt8 srcMac, myMac;
    uInt1 sid, hops, dHops, sSpan, dSpan, best, lost, sEnd, dEnd, dsid, i;

    GetArgs(dPtr,fPtr,&sid,&hops,&sSpan,&dSpan,&best,&lost,&srcMac,&myMac);
    if (hops>SMAX)
        return;
    dsid= dPtr->sidInfo[hops].dsid;
    dHops= dPtr->hopInfo[sid].sHops;
    if (myMac==srcMacMac) {
        dPtr->c= 0;                                                 // Self makes chain
        dEnd= sEnd= hops;                                           //  w/symmetric lists
    } else if (fPtr->protect) {
        sEnd= (dPtr->c==0) ? sEnd-hops:sSpan;                       // Sometimes 2 breaks
        dPtr->c= 1;                                                 // Loop is broken w/
        dEnd= hops;                                                 // protection hops
    } else {
        sEnd= sSpan;                                                // Span remains the
        dEnd= dSpan;                                                //  same by default
    }
    for (sPtr= &(dPtr->sidInfo[i=sEnd+1]); i<=sSpan; i+=1,sPtr+=1)
        sPtr->s= 0, sPtr->ssid= SMAX;                               // Src-side reduction
    for (sPtr= &(dPtr->sidInfo[i=dEnd+1]); i<=dSpan; i+=1,sPtr+=1)
        sPtr->d= 0, sPtr->dsid= SMAX;                               // Dst-side reduction

    dPtr->sSpan= sEnd;                                              // Set src-side span
    dPtr->dSpan= dEnd;                                              // Set dst-side span
    dPtr->sidInfo[hops].dsid= sid;                                  // Set src-side SID
    dPtr->hopInfo[sid].dHops= hops;                                 // Set src-side hops
    dPtr->hopInfo[sid].sourceMacAddress= best ? myMac:srcMac;       // Dup MAC overwrite
    if (lost)
        dPtr->nsid= ((dPtr->msid=dPtr->nsid)+1)%SMAX;               // Next available sid
    if (best!=0||lost!=0||sEnd!=sSpan||dEnd!=dSpan||sid!=dsid||hops!=dHops)
        for (i=1; i<SIDS; i+= 1)  dPtr->sidInfo[i].d= (i<=next);    // Change => queries
    dPtr->sidInfo[hops].d= 0;                                       // Cancel this query
    dPtr->q|= fPtr->q;                                              // but catch others
}
```

F.10 Figures for Bob

Simplified RPR MAC data-path model, as shown in Figure F.40.

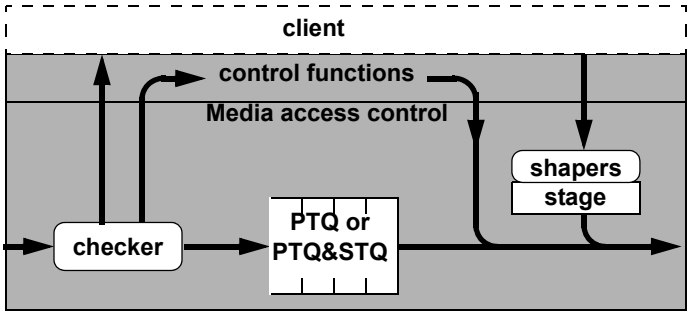


Figure F.40—Simplified RPR MAC transit path

Another figure.

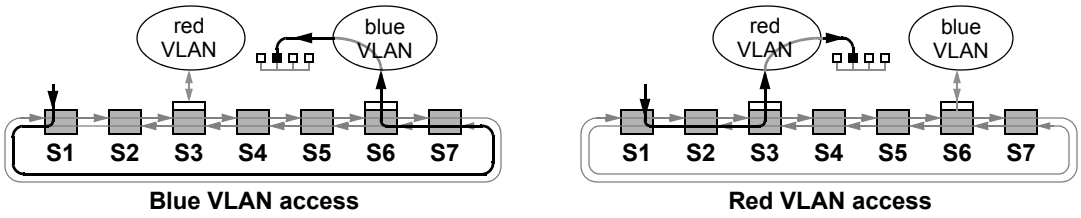


Figure F.41—Spatial reuse of independent VLANs

Annex G.

(normative)

Time-of-day distribution

G.1 Time-of-day synchronization

Time-of-day synchronization involves the tight synchronization of timers maintained on clock-master and clock-slave stations. The intent is to provide uniform stratum clocks, to enable synchronization of source and destination devices, to avoid data slips or gaps during the distribution and/or presentation of real-time information, such as telephony traffic.

Some physical layers, such as SONET provide stratum clock services. This subannex describes how these services may be provided at higher layers within other less-supportive physical layers.

G.1.1 Wallclock synchronization

With bidirectional1 cables, the clockSync transmissions can account for the constant cable-induced delays, by measuring round-trip cable delays. Using such techniques, the accuracy of these wallclock synchronization protocols is dependent on the delay differences between incoming and outgoing links, not the overall delay of either. Implementation of these wallclock synchronization protocols involves monitoring the arrival and departure of clockSync frames.

As an example, consider the synchronization of stationY to the time reference supplied by station-X. Both stations have free-ringletning timers *timerAx* and *timerAy*, as illustrated in Figure G.42. These timers are added to offset values *offsetAx* and *offsetAy* to generate stationX and stationY time references respectively. Synchronization of stationY involves the continuous adjustment of *offsetAy*, with the intent of causing stationY's *timerAy+offsetAy* sum to track stationX's *timerAx+offsetAx* sum.

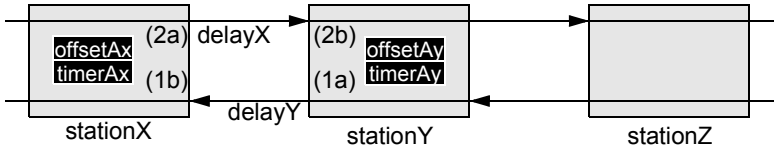


Figure G.42—Time synchronization sequences

Synchronization involves periodic stationY-to-stationX transmission of clockSync frames. The transmitting stationY is responsible for accurately recording (1a) the clockSync transmission time; the recieving station is responsible for accurately recording (1b) the clockSync reception time. StationY is responsible for placing the recorded clockSync transmission time in the next clockSync frame sent to stationX.

At stationX, the value of *deltaY* is generated by subtracting stationX's observed clockSync-recieve time from stationY's communicated clockSync-transmit time (this approximates *delayY+timerAy–timerAx)*. The *deltaY* value is communicated to stationY by piggy-backing on the next stationX-to-stationY clockSync transmission.

Synchronization also involves concurrent periodic stationX-to-stationY transmission of *clockSync* frames. The transmitting stationX is responsible for accurately recording (2a) the clockSync transmission time; the

recieving station is responsible for accurately recording (2b) the clockSync reception time. StationX is responsible for placing the recorded clockSync transmission time in the next clockSync frame sent to stationY.

At stationY, the value of *deltaX* is generated by subtracting stationX's observed clockSync-recieve time from stationY's communicated clockSync-transmit time (this approximates *delayX+timerAx–timerAy)*. Synchronization of stationY is based on the assumption that *delayX* and *delayY* values are equal, setting *offsetAy=offsetAx+(deltaX–deltaY)/2*.

G.1.2 Synchronization state

The root station is responsible for generation of clockSync frames. All stations (root as well as nonroot) are responsible for measuring the clockSync propagation time through themselves. Clock deviations are sampled in cycle N and calibrations are performed in cycle N+1. Clock sampling involves through-station delays measurements and sampling of the station's *clockTime* value at its transmitter, as illustrated in Figure G.43**.**

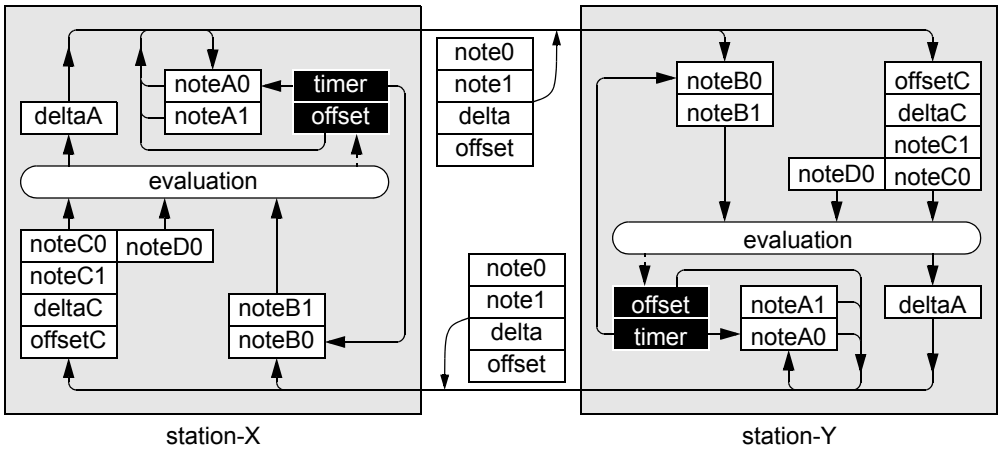


Figure G.43—Time synchronization components

The behavior on synchronization protocols is based on the following behavior of each station:

a) At periodic intervals, a clockSync frame is sent to the neighbor station:
   1) The values of *noteA0*, *noteA1*, *offset*, and *delta* are placed in this clockSync frame.
   2) At the start of this clockSync-frame transmission, registers are updated as follows:
      i) Set *noteA1=noteA0*, so past snapshot time can be sent in the next clockSync frame.
      ii) Set *noteA0=timer*, so this snapshot time can be sent in the next clockSync frame.

b) When a clockSync frame is first received from the neighbor station:
   1) At the start of this clockSync-frame reception, registers are updated as follows:
      i) Set *noteB1=noteB0*, so past snapshot time can be remembered.
      ii) Set *noteB0=timer*, so that *noteB1* can be produced in the future.
   2) Set *noteD0=noteC0*, so the integrity of the next clockSync frame can be verified.
   3) Save clockSync's *note0*, *note1*, and *delta* contents in *noteC0*, *noteC1*, and *deltaB* respectively.

c) The following computations are performed if *noteC1==noteD0*, after clockSync-frame arrival:
   1) Compute and set *deltaA=noteC0-noteB1*, so this time measurement can be returned.
   2) If synchronizing to the adjacent neighbor, set *offset=offsetC+(deltaA–deltaC)/2*.