Proposed Text for
A Reworked P802.1Qdd RAP Propagator

Individual Contribution by

Feng Chen

(Siemens AG)

Johannes Specht

(Self, Siemens AG)

2022-02-21

1 **Table of Contents**

2 1 Introduction 4
3 1.1 Overview 4
4 1.2 Conventions 4
5 1.2.1 State Machine Diagrams 4
6 1.2.2 Pseud-Code Presentation 5
7 1.2.3 Variables and Scopes 5
8 1.2.4 create, delete, NULL 5
9 99.x Interfaces 5
10 99.x.1 RAP Participant service primitives (99.6.4.2) 5
11 99.x.1.1 DECLARE_ATTRIBUTE.request(portID, attr) 5
12 99.x.1.2 WITHDRAW_ATTRIBUTE.request(portID, attr) 5
13 99.x.1.3 ATTRIBUTE_REGISTERED.indication(portID, attr) 5
14 99.x.1.4 ATTRIBUTE_DEREGISTERED.indication(portID, attr) 5
15 99.x.1.5 DECLARATION_OPER_STATE.indication(portID, operState) 5
16 99.x.2 Global service primitives 6
17 99.x.2.1 VLAN_CONTEXT_TOPO_CHANGE.indication(vid) 6
18 99.x.2.2 STREAM_DA_REGISTERED.indication(portID, macAddr) 6
19 99.x.2.3 STREAM_DA_DEREGISTERED.indication(portID, macAddr) 6
20 99.7 RAP Propagator 6
21 99.7.1 RAP Propagator overview 6
22 99.7.2 RAP Propagator state machine diagrams 6
23 99.7.3 RAP Propagator variables 14
24 99.7.3.1 taReg 14
25 99.7.3.2 taDec 14
26 99.7.3.3 laReg 14
27 99.7.3.4 laDec 15
28 99.7.3.5 localRaClass 15
29 99.7.3.6 neighborRaClass 15
30 99.7.3.7 port 15
31 99.7.3.8 portRaClass 15
32 99.7.4 RAP Propagator procedures 16
33 99.7.4.1 validateTaReg(ePortID, eTaAttr) 16
34 99.7.4.2 processTaIngress(eTaReg) 16
35 99.7.4.3 processTaEgress(eTaDec) 16
36 99.7.4.4 processLaIngress(eLaReg) 17

1 99.7.4.5 processLaEgress(eTaReg) 17
2 99.7.4.6 getTaRegDestPortIDs(eTaReg) 18
3 99.7.4.7 getTaStatus(eTaAttr) 18
4 99.7.4.8 failTa(eTaAttr, eFailureCode) 18
5 99.7.4.9 constructLaAttr(eStreamID, eVID, eLaStatus) 19
6 99.7.4.10 constructRaAttr() 19
7 99.7.4.11 getLocalRaClass(ePriority) 19
8 99.7.4.12 setDomainBoundaryStatus(ePortID) 19
9 99.7.4.14 checkResources(eTaDec) 20
10 99.7.4.15 adjustTa(eTaDec) 20
11 99.7.4.16 allocateResources(eTaDec) 20
12 99.7.4.17 deallocateResources(eTaDec) 20
13 (Annex Z) Collected Issues during Development of this Document 21
14 Z.1 Camel-Case vs. Underscore-Case 21
15 Z.2 VLAN-aware LA attribute 21
16 99.4.4 Listener Attach attribute and TLV encoding 21
17 99.4.4.1 StreamID 21
18 99.4.4.2 VID 21
19 99.4.4.3 ListenerAttachStatus 21
20 Z.4 Sufficient validation of Talker Announce Registrations 22
21 Z.5 Issues related to RA class registrations 23
22
23

1 Introduction

1.1 Overview

This document contains a re-worked version of the RAP Propagator defined in subclause 99.7 of IEEE P802.1Qdd draft D0.5. Based on review feedback, several aspects were identified where clarity and readability of the technical contents defined so far in D0.5 can be enhanced.

The enhancements in the re-worked version include the following:

- New subclause structure, simplifying finding particular contents
- Contents ordered for top-down reading
- Removal of several synonyms/ambiguously used terms ("functions" vs. "procedures", etc.)
- More formal description of protocol mechanisms
- Removal of unnecessary concurrency
- Various enhancements in non-formal descriptions (including shortening)

The re-worked contents are intended for incorporation into the next Qdd draft D0.6.

In this re-worked version, the functionality and operation of RAP Propagator is specified in terms of state machine diagrams, along with their variables and procedures. As an effective means of enhancing clarity and readability, pseudo-code has been used in describing the actions to be taken by a state or a procedure. The conventions for the descriptive methods used by this document are stated in 1.2.

In addition, several potential technical corrections and enhancements have been also identified during development of this document and collected in Annex Z at the end of this document. These technical issues need to be considered and discussed in the development of future Qdd drafts.

The remainder of this document is structured as follows:

- Clause 1.2 defines conventions for the proposed new representation.
- Clause 99.x summarizes the external interfaces between RAP Propagator and other entities (e.g., RAP Participant). A subset of these interfaces is found in P802.1/D0.5, in which case references are provided for further description.
- Clause 99.7 contains the re-worked version of the RAP Propagator.

1.2 Conventions

1.2.1 State Machine Diagrams

This document used state machine diagrams that contain pseudo-code when considered appropriate by the authors. These diagrams are aligned to the conventions in Annex E of IEEE Std 802.1Q-2018 to a large extent. The following extensions apply:

a) All transitions and operations are atomic and finish without any progress of time, as soon as the associated conditional expressions(s) of transition(s) is/are satisfied. The sole reason for steady states is that none of the conditional expressions of any outgoing transition of a particular state is satisfied.

b) Service primitives can occur in the conditional expressions of transitions. Invocation of such service primitives immediately leads to activation of a transition, provided that:

   1) All other sub-expressions are satisfied, if such expressions exist, and

   2) The state machine resides in the associated state prior to invocation.

c) Service primitives that cannot be processed immediately are queued in their order of invocation for subsequent processing (i.e., no service primitive is lost).

d) In case the conditional expressions of more than one transition are satisfied simultaneously, the taken transition is arbitrarily.

1.2.2 Pseud-Code Presentation

The pseudo-code found in this document is inspired by C++. The emphasis is on simplicity and clarity of specification and unambiguous description of the externally visible behavior. Efficiency (speed, memory usage, etc.) is left to the implementation (in software/firmware, hardware, combinations of the aforesaid, etc.).

1.2.3 Variables and Scopes

The RAP operation relies on several Bridge-local variables on different scopes, such as per-Bridge, per-Bridge per Port, or per-Bridge per Port per Stream. The pseudo-code uses an array notation (i.e., identifier followed by index expressions in brackets) to express such scopes, in which the array index expressions are used for addressing one or more entries in such arrays.

For example,

xyz[*<portID>*, *<StreamID>*] and **xyz**[*<portID>*][*<StreamID>*]

are semantically identical and address an entry in an array **xyz** with index expressions for given Stream ID (*<StreamID>*) and a given Port ID (*<portID>*).

The asterisk character (‘*’) is used as a wildcard for index expressions to address all items in the respective array dimension.

1.2.4 create, delete, NULL

Lifecycle management of entries in array variables in deeper Bridge-local scopes is realized by the two functions **create** and **delete**; such variables is initialized to special value **NULL**. The semantics of the two functions is as follows:

a) Invocation of **create** *<identifier>*[…] allocates an entry in an array variable and returns a reference, provided that the variable is **NULL** prior to this invocation. If it is not **NULL**, prior to the invocation of **create**, the existing reference is returned.

b) Invocation of **delete** *<identifier>*[…] deallocates an entry in an array variable and sets it back to special value **NULL**, provided that the variable is other than **NULL** prior to this invocation. The variable remains **NULL** if it had this value prior to the invocation of **delete**.

99.x Interfaces

99.x.1 RAP Participant service primitives (99.6.4.2)

<< Note: This clause is provided in this document to provide references to P802.1Qdd/D0.5 and introduce a new per-RAP participant primitive DECLARATION_OPER_STATE.indication(portID, operState) in 99.x.1.5 not present in P802.1Qdd/D0.5.>>

99.x.1.1 DECLARE_ATTRIBUTE.request(portID, attr)

See 99.6.4.2.1 of P802.1Qdd/D0.5.

99.x.1.2 WITHDRAW_ATTRIBUTE.request(portID, attr)

See 99.6.4.2.2 of P802.1Qdd/D0.5.

99.x.1.3 ATTRIBUTE_REGISTERED.indication(portID, attr)

See 99.6.4.2.3 of P802.1Qdd/D0.5.

99.x.1.4 ATTRIBUTE_DEREGISTERED.indication(portID, attr)

See 99.6.4.2.4 of P802.1Qdd/D0.5.

99.x.1.5 DECLARATION_OPER_STATE.indication(portID, operState)

This primitive is used by a RAP Participant on a Port (portID) to notify the service user of the operational state of its declaration function. The operState parameter contains a Boolean value that indicates whether the RAP Participant’s declaration function is operational (TRUE) or not (FALSE).

<< Note: This primitive is introduced to indicate a change to the value of portalCreated variable (99.6.2.2) in a RAP Participant.>>

99.x.2 Global service primitives

<< Note: The following primitives are invoked by the local bridge in the events that could affect the current results of TA propagation, to signal the RAP Propagator that there is a need to reprocess propagation of the TA registrations. >>

99.x.2.1 VLAN_CONTEXT_TOPO_CHANGE.indication(vid)

This indication is invoked by the local Bridge to indicate a change in the set of Ports that forms the topology of a given VLAN context (99.2.4.1) identified by vid. Such a change can result from a spanning tree (7.3) or VLAN membership (7.4) reconfiguration.

99.x.2.2 STREAM_DA_REGISTERED.indication(portID, macAddr)

This indication is invoked by the local Bridge to indicate that a destination MAC address (macAddr) is registered on a Port (portID).

99.x.2.3 STREAM_DA_DEREGISTERED.indication(portID, macAddr)

This indication is invoked by the local Bridge to indicate that a destination MAC address (macAddr) is deregistered on a Port (portID).

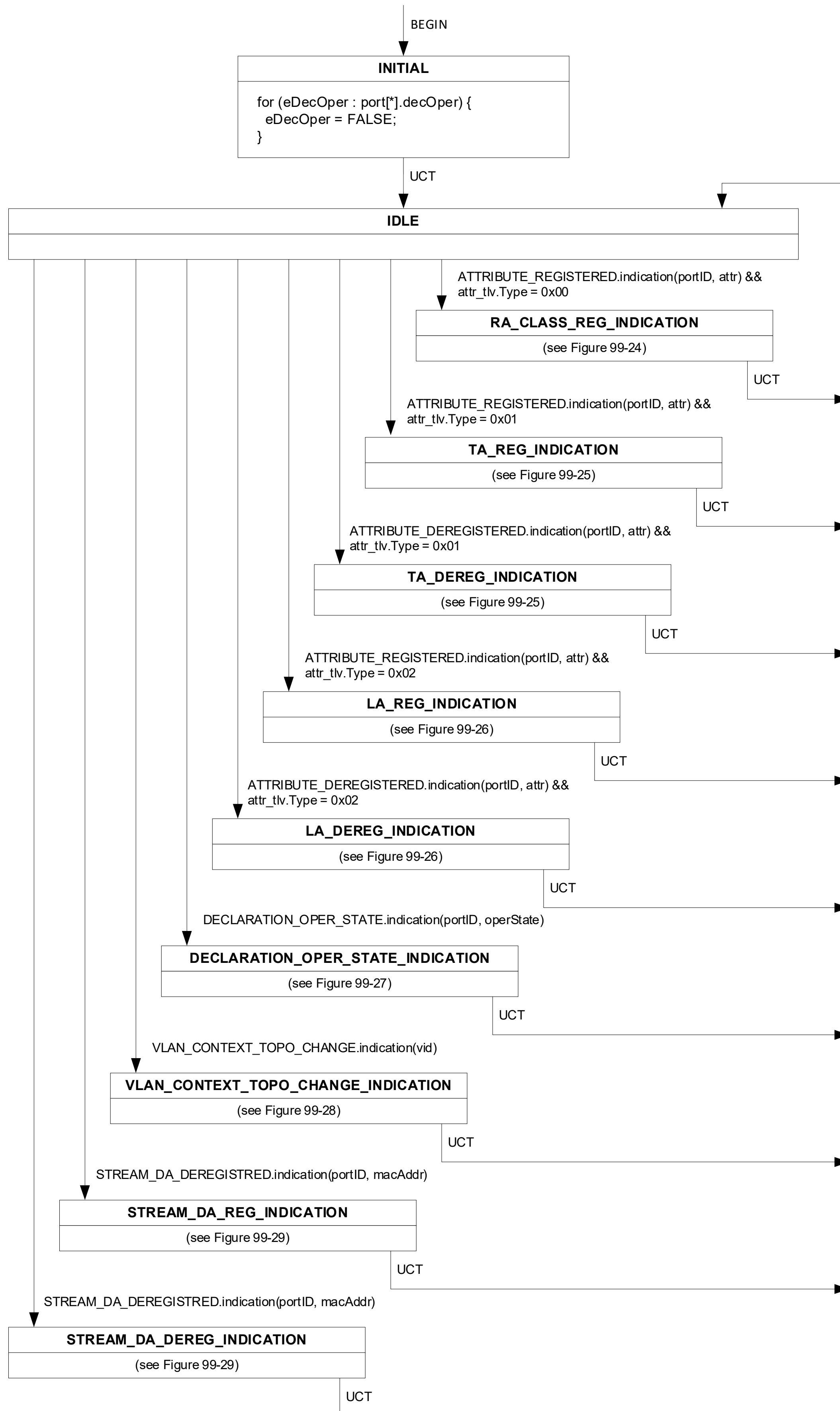
99.7 RAP Propagator

99.7.1 RAP Propagator overview

The operation of the RAP Propagator is defined by the state machine diagrams in 99.7.2 following the conventions in 1.2. Transitions in these diagrams are initiated by the events summarized in 99.7.3. The operations in the state machine diagrams operates on the variables specified in 99.7.4 and utilizes the procedures in 99.7.5.

99.7.2 RAP Propagator state machine diagrams

The operation of a RAP Propagator is defined by the state machine in Figure 99-23, where the procedures defined for the states (with the exception of the INITIAL and IDLE states) are illustrated in Figure 99-24 through Figure 99-29.



1

2 **Figure 99-23— RAP Propagator state machine**

The processing of RA class registration is defined by the state machine diagram in Figure 99-24.

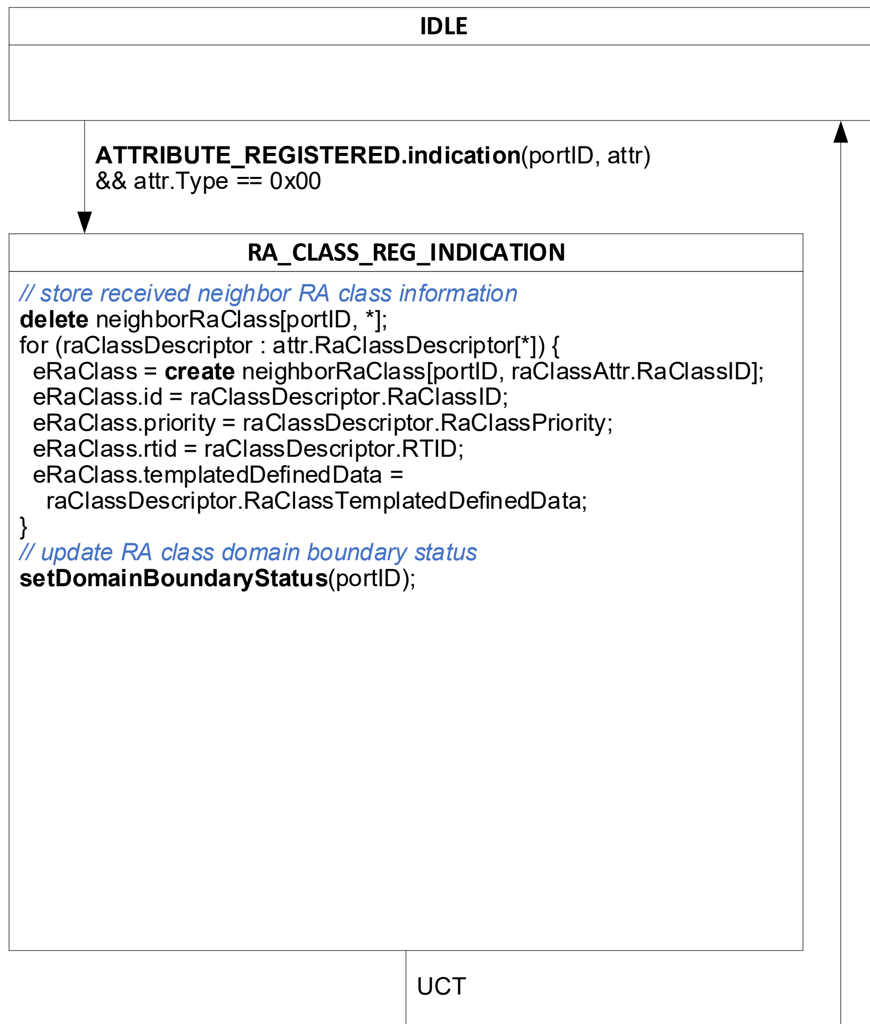


Figure 99-24— Processing of RA class registration

The processing of Talker Announce registration and deregistration is defined by the state machine diagram in Figure 99-25.

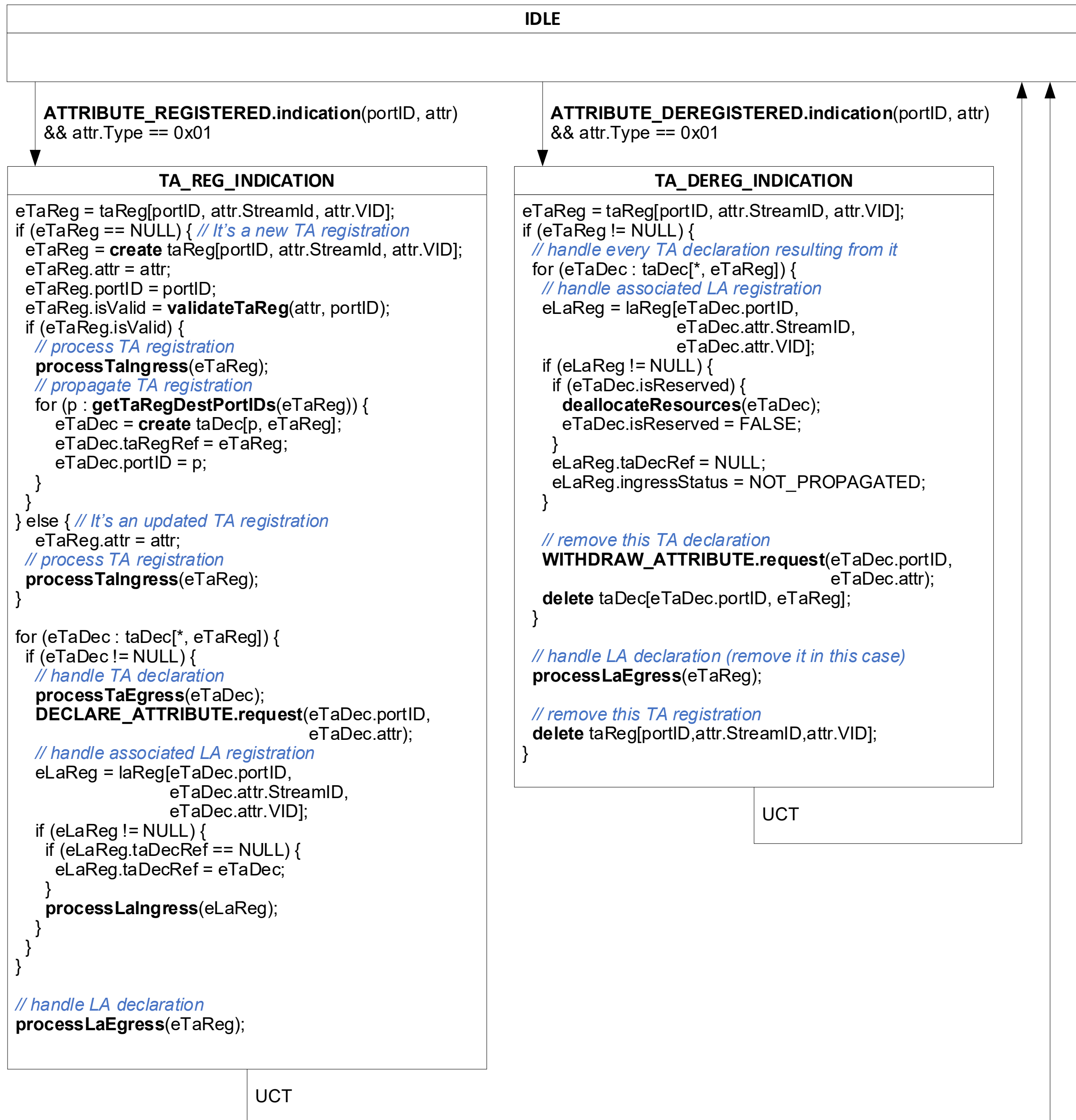


Figure 99-25— Processing of Talker Announce registration and deregistration

The processing of Listener Announce registration and deregistration is defined by the state machine diagram in Figure 99-26.

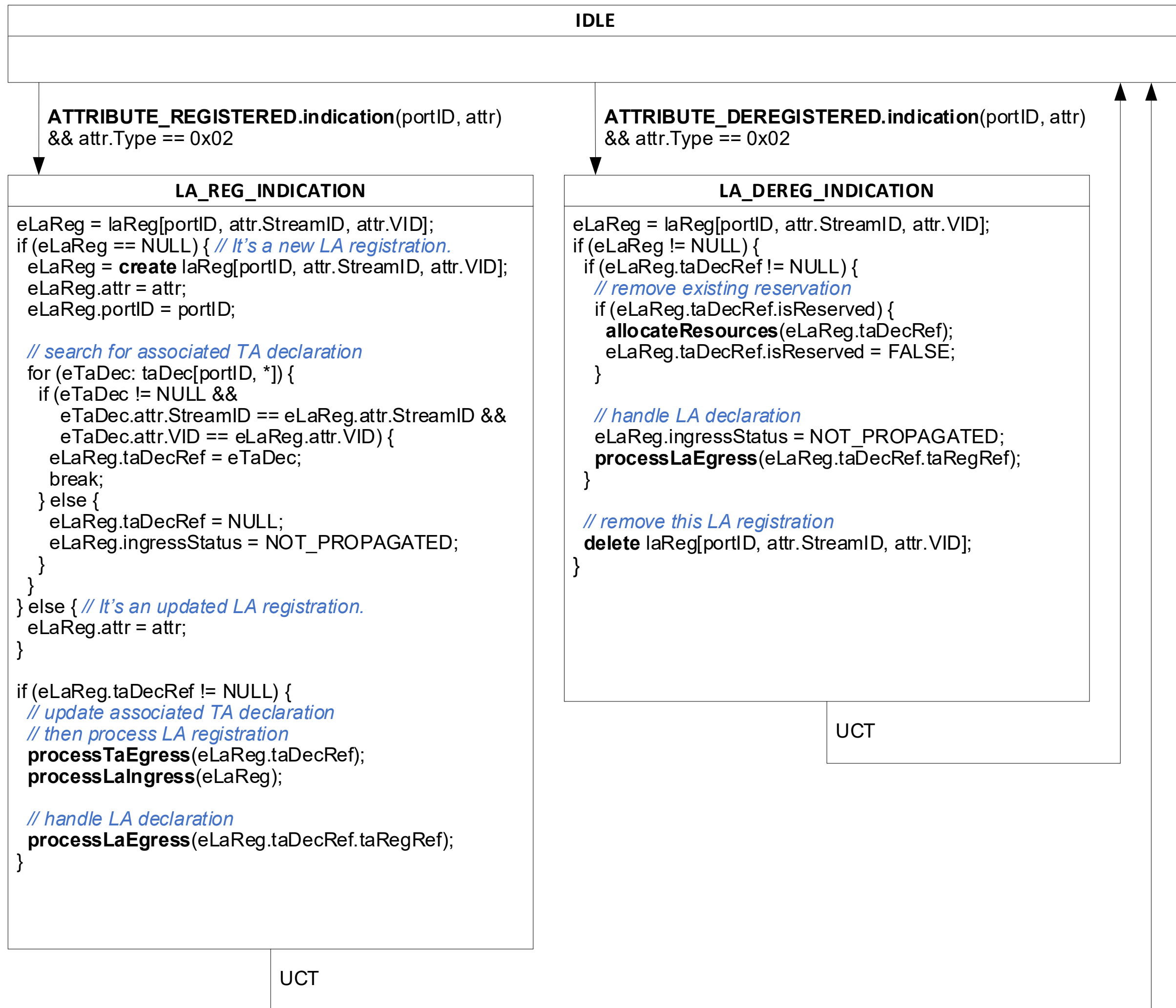


Figure 99-26 — Processing of Listener Attach registrations and deregistration

The processing of operational state change indications of the declaration function of RAP participants is defined in Figure 99-27.

Figure 99-27 — Processing of operational state change indications of the declaration function of RAP participants

The processing of VLAN context topology change indications is define by the state machine diagram in Figure 99-28.

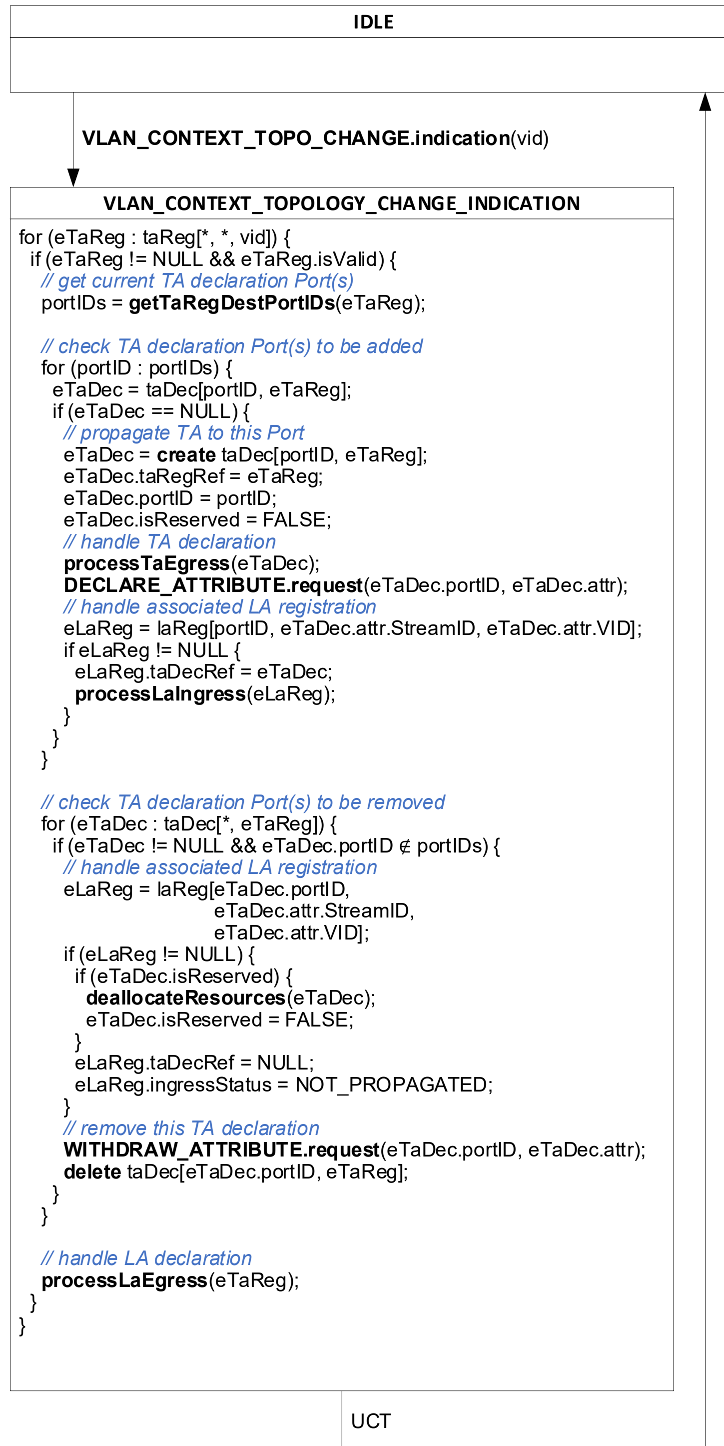


Figure 99-28 — Processing of VLAN context topology change indications

The processing of Stream Destination Address registration (99.x.2.2) and deregistration (99.x.2.3), respectively, is defined by the state machine diagram in Figure 99-29.

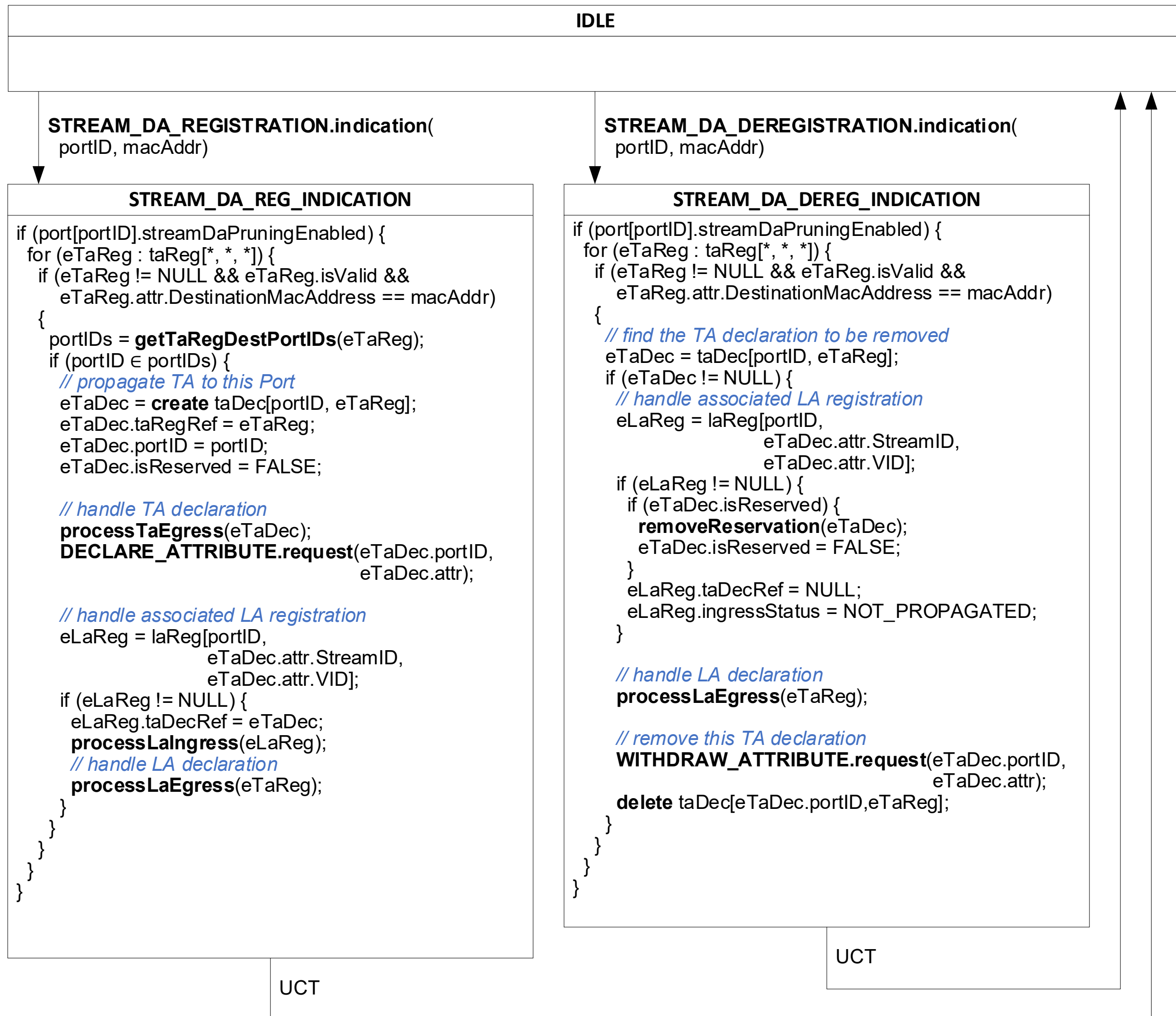


Figure 99-29 — Processing of Stream Destination Address registration and deregistration

99.7.3 RAP Propagator variables

99.7.3.1 taReg

The taReg variable is a 3-dimensional array, indexed by ingress Port, Stream ID (99.4.3.1) and VID (99.4.3.5.3), that contains entries representing Talker Announce registrations. Entries in taReg are records comprising the following elements:

a) **attr**: A registered Talker Announce attribute (99.4.3).

b) **portID**: The ingress Port on which the attribute in item a) is registered.

c) **isValid**: A Boolean value indicating whether the Talker Announce registration contained in this entry is valid (TRUE) or not (FALSE), as determined by the validateTaReg() procedure (99.7.4.1).

d) **ingressStatus**: The status of this Talker Announce registration on ingress determined by the processTaIngress() procedure (99.7.4.2), taking one of the following enumerated values:

   1) **TA_RECV_FAIL**: This Talker Announce registration contains a failure code generated by an upstream station.

   2) **TA_INGRESS_SUCCESS**: This Talker Announce registration contains no failure code and is not failed on ingress of this Bridge.

   3) **TA_INGRESS_FAIL**: This Talker Announce registration contains no failure code but failed on ingress of this Bridge with a failure code contained in ingressFailureCode [item e), below].

e) **ingressFailureCode**: A RAP failure code.

<< Note: Item e) should be further detailed and discussed in D0.6 or later.>>

99.7.3.2 taDec

The taDec variable is a 2-dimensional array, indexed by egress Port and taReg entry, that contains entries representing Talker Announce declarations. Entries in taDec are records comprising the following elements:

a) **taRegRef**: A reference to a taReg entry from which this taDec entry results. The value of taRegRef is identical to taReg entry index of the taDec variable.

b) **attr**: A Talker Announce attribute determined by the processTaEgress() procedure (99.7.4.3) for declaration on the associated egress Port.

c) **portID**: The egress Port on which the Talker Announce declaration is made.

d) **isReserved**: A Boolean value indicating whether a reservation has been made for this TA declaration (TRUE) or not (FALSE).

99.7.3.3 laReg

The laReg variable is 3-dimensional array, indexed by ingress Port, Stream ID (99.4.4.1) and VID (99.4.4.2), that contains entries representing Listener Attach registrations. Entries in laReg are records comprising the following elements:

a) **attr**: A registered Listener Attach attribute (99.4.3).

b) **portID**: The Port on which the attribute in item a) of this entry is registered.

c) **taDecRef**: A NULL value indicating this Listener Attach registration has no associated Talker Announce declaration, or a reference to a taDec entry in the taReg that contains the Talker Announce declaration with which this Listener Attach registration is associated.

d) **ingressStatus**: The status of this Listener Attach registration on ingress determined by the processLaIngress() procedure (99.7.4.4), taking one of the following enumerated values:

   1) **NOT_PROPAGATED**: This Listener Attach registration has no associated Talker Announce declaration and is not propagated.

2) **ATTACH_READY**: This Listener Attach registration has an associated Talker Announce declaration and is propagated with the Attach Ready status [item a) in 99.2.5.2].

3) **ATTACH_FAIL**: This Listener Attach registration has an associated Talker Announce declaration and is propagated with the Attach Fail status [item b) in 99.2.5.2].

4) **ATTACH_PARTIAL_FAIL**: This Listener Attach registration has an associated Talker Announce declaration and is propagated with the Attach Partial Fail status [item c) in 99.2.5.2].

99.7.3.4 laDec

The laDec variable is a one-dimensional array, indexed by taReg entry, that contains entries representing Listener Attach declarations. Entries in laDec are records comprising the following elements:

a) **taRegRef**: A reference to a taReg entry with which this laDec entry is associated.

b) **attr**: A Listener Attach attribute determined by the processLaEgress() procedure (99.7.4.5) for declaration on the associated egress Port (i.e., taRegRef.portID).

99.7.3.5 localRaClass

The localRaClass variable is a one-dimensional array, indexed by RA class ID (99.2.2.1), that contains entries describing local RA classes. Entries in localRaClass are records comprising the following elements:

a) **id**: A 8-bit RA class ID (99.2.2.1).

b) **priority**: A 8-bit RA class priority (99.2.2.2)

c) **rtid**: A 32-bit RTID (99.2.2.3)

d) **templateDefinedData**: An octet string, possibly zero length, as the value to be carried in the RA Class Template Defined Data field (99.4.2.1.4) of the RA Class Descriptor sub- TLV for this RA class in an RA Class attribute declared by the RAP Propagator.

99.7.3.6 neighborRaClass

The neighborRaClass variable is a two-dimensional array, indexed by Port and RA class ID (99.2.2.1), that contains entries storing RA class descriptions received from neighbor stations. Entries in neighborRaClass are records comprising the following elements:

a) **id**: A 8-bit RA class ID (99.2.2.1).

b) **priority**: A 8-bit RA class priority (99.2.2.2).

c) **rtid**: A 32-bit RTID (99.2.2.3)

d) **templateDefinedData**: An octet string, possibly zero length, containing the value carried in the RA Class Template Defined Data field (99.4.2.1.4) of the RA Class Descriptor sub-TLV for this RA class in the RA class attribute registered on the Port.

99.7.3.7 port

The port variable is a one-dimensional array, indexed by Port, that contains entries for controlling the operation of the RAP Propagator on a per Port basis. Entries in port are records comprising the following elements:

a) **streamDaPruningEnabled:** A Boolean indicating whether Stream DA Pruning (99.2.4.2) on a given Port is administratively enabled (TRUE) or disabled (FALSE). The default value is FALSE.

b) **decOper:** A Boolean indicating whether the attribute declaration function on a Port is operational (TRUE) or not (FALSE). The default value is FALSE.

99.7.3.8 portRaClass

The portRaClass variable is two-dimensional array, indexed by Port and RA Class ID (99.2.2.1), that contains entries for controlling the operation of the RAP Propagator on a per Port, per RA class basis. Entries in portRaClass are records comprising the following elements:

a) **domainBoundaryStatus:** A Boolean indicating whether a Port is a domain boundary port (99.2.2.4) for an RA class (TRUE) or not (FALSE).

b) **maxFrameSize:** A 16-bit unsigned integer, indicating the maximum frame size, in octets, of the streams that can be reserved in an RA class on a Port.

c) **minFrameSize:** A 16-bit unsigned integer, indicating the minimum frame size, in octets, of the streams that can be reserved in an RA class on a Port.

d) **maxBandwidth:** A 32-bit unsigned integer, indicating the maximum amount of bandwidth that can be reserved for use by an RA class on a Port. The bandwidth value is represented as a percentage of the Port's transmission rate determined by the operation of the underlying MAC and expressed as a fixed-point number scaled by a factor of 1,000,000; i.e., 100,000,000 (the maximum value) represents 100%.

e) **maxHopDelay:** A 32-bit unsigned integer, indicating the maximum delay, in nanoseconds, provided by an RA class for all the streams that are reserved with that RA class on a Port.

99.7.4 RAP Propagator procedures

99.7.4.1 validateTaReg(ePortID, eTaAttr)

This procedure determines whether a Talker Announce registration is valid or not.

<< Note: The operation of this procedure appears non-trivial. Further discussion and some text for later technical consideration is found in Annex Z.4.>>

99.7.4.2 processTaIngress(eTaReg)

This procedure performs ingress processing for a Talker Announce registration in a taReg entry (eTaReg), as follows:

```
processTaIngress(eTaReg) {
  if (getTaStatus(eTaReg.attr) == Announce Fail) {
    // TA failed by an upstream station
    eTaReg.ingressStatus = TA_RECV_FAIL;
  } else {
    // find the local RA class for this TA
    eLocalRaClass = getLocalRaClass(eTaReg.attr.Priority);

    // find the neighbor RA class for this TA
    for (eNeighborRaClass : neighborRaClass[eTaReg.portID,*]) {
      if (eNeighborRaClass != NULL &&
          eNeighborRaClass.priority == eTaReg.attr.Priority)
      {
        break;
      } else {
        eNeighborRaClass = NULL;
      }
    }

    if (eLocalRaClass    == NULL ||
        // this TA is not associated with any local RA class
        eNeighborRaClass == NULL ||
        // this TA is not associated with any neighbor RA class
        eLocalRaClass.id != eNeighborRaClass.id)
        // unmatching RA class ID between local and neighbor RA class
    {
      // TA registration is received across an RA class domain boundary
      eTaReg.ingressStatus = TA_INGRESS_FAIL;
      eTaReg.ingressFailureCode = << Failure code TBD >>;
    } else {
      // TA registration is received within an RA class domain
      eTaReg.ingressStatus = TA_INGRESS_SUCCESS;
    }
  }
}
```

99.7.4.3 processTaEgress(eTaDec)

This procedure performs egress processing for a Talker Announce declaration in a taDec entry (eTaDec), as follows:

```
processTaEgress(eTaDec) {
  eTaReg = eTaDec.taRegRef;
  if (eTaReg.ingressStatus == TA_INGRESS_SUCCESS){
    // TA is failed neither by an upstream station nor on ingress of this Bridge
    if (!eTaDec.isReserved) {
      // the stream not yet reserved -> check resources
      resFailCode = checkResources(eTaDec);
      if (resFailCode == 0) {
        // the stream is reservable (resource check ok).
        // update accumulated latency, Tspec.
        eTaDec.attr = adjustTa(eTaDec);
      } else {
        // the stream is unreservable (resource check failed).
        // declare with the egress failure code.
        eTaDec.attr = failTa(eTaReg.attr, resFailCode);
      }
    } else {
      // the stream already reserved -> resource checking once again not needed.
      // eTaDec.attr already contains attribute previously adjusted for declaration.
    }
  } else if (eTaReg.ingressStatus == TA_INGRESS_FAIL) {
    // TA is failed on ingress of this Bridge: declare with the ingress failure code
    eTaDec.attr = failTa(eTaReg.attr, eTaReg.ingressResFailCode);

  } else if (eTaReg.ingressStatus == TA_RECV_FAIL) {
    // This TA registration is failed by an upstream Bridge -> declare as-is.
    eTaDec.attr = eTaReg.attr;
  }
}
```

99.7.4.4 processLaIngress(eLaReg)

This procedure performs ingress processing for a Listener Attach registration in a laReg entry (eLaReg) that is associated with a Talker Announce declaration (eLaReg.taDecRef), as follows:

```
processLaIngress(eLaReg) {
  if (eLaReg.attr.ListenerAttachStatus  == Attach Fail ||
      getTaStatus(eLaReg.taDecRef.attr) == Announce Fail)
  {
    // reservation not allowed
    if (eLaReg.taDecRef.isReserved == TRUE) {
      // remove the existing reservation
      deallocateResources(eLaReg.taDecRef);
      eLaReg.taDecRef.isReserved = FALSE;
    }
    // propagate LA as Attach Fail
    eLaReg.ingressStatus = ATTACH_FAIL;
  } else {
    // current LA status is Attach Ready or Attach Partial Fail,
    // and current TA declaration status is Announce Success, reservation allowed.
    if (!eLaReg.taDecRef.isReserved) {
      allocateResources(eLaReg.taDecRef);
      eLaReg.taDecRef.isReserved = TRUE;
    }
    // propagate LA as-is, which is either Attach Ready or Attach Partial Fail
    eLaReg.ingressStatus = eLaReg.attr.ListenerAttachStatus;
  }
}
```

99.7.4.5 processLaEgress(eTaReg)

This procedure performs egress processing for a Listener Attach declaration in a laDec entry that is associated with the given taReg entry (eTaReg), as follows:

```
processLaEgress(eTaReg) {
  // collect statistics from all LA registrations to be merged into this LA declaration
  numPropagated = numReady = numFailed = 0;
  for (eLaReg : laReg[*, eTaReg.attr.StreamID, *]) {
```

```
1       if (eLaReg != NULL &&
2           eLaReg.taDecRef != NULL &&
3           eLaReg.taDecRef.taRegRef == eTaReg &&
4           eLaReg.ingressStatus != NOT_PROPAGATED)
5       {
6         numPropagated++;
7         if      (eLaReg.ingressStatus == ATTACH_READY) numReady++;
8         else if (eLaReg.ingressStatus == ATTACH_FAIL)  numFailed++;
9       }
10    }
11    // determine LA declaration status (LA merging)
12    eLaDec = laDec[eTaReg];
13    if (numPropagated > 0) {
14      // at least one LA registration is propagated
15      if (eLaDec == NULL) {
16        eLaDec = create laDec[eTaReg];
17        eLaDec.taRegRef = eTaReg;
18      }
19      eStreamID = eTaReg.attr.StreamID;
20      eVID = eTaReg.attr.VID;
21      if (numPropagated == numReady) {
22        // all propagated as Attach Ready -> declare Attach Ready
23        eLaDec.attr = constructLaAttr(eStreamID, eVID, Attach Ready);
24      } else if (numPropagated == numFailed) {
25        // all propagated as Attach Fail -> declare Attach Fail
26        eLaDec.attr = constructLaAttr(eStreamID, eVID, Attach Fail);
27      } else {
28        // Otherwise: declare Attach Partial Fail
29        eLaDec.attr = constructLaAttr(eStreamID, eVID, Attach Partial Fail);
30      }
31      DECLARE_ATTRIBUTE.request(eLaDec.taRegRef.portID, eLaDec.attr);
32    } else {
33      // no LA registration is propagated
34      // withdraw current LA declaration if existing
35      if eLaDec != NULL {
36        WITHDRAW_ATTRIBUTE.request(eLaDec.taRegRef.portID, eLaDec.attr);
37        delete laDec[eTaReg];
38    }
39  }
```

40 **99.7.4.6 getTaRegDestPortIDs(eTaReg)**

41 This procedure determines the Port(s) to which the Talker Announce registration contained in a taReg entry
42 (eTaReg) is to be propagated. It returns a set of portIDs, possibly empty, each of which meets all the following
43 conditions:

44 a) portID != eTaReg.portID;

45 b) port[portID].decOper == TRUE;

46 c) portID is in the VLAN context identified by eTaReg.attr.VID, in accordance with 99.2.4.1;

47 d) If port[portID].streamDaPruningEnabled == TRUE, eTaReg.attr.DestinationMacAddress is registered on
48 the Port with portID, in accordance with 99.2.4.2.

49 **99.7.4.7 getTaStatus(eTaAttr)**

50 This procedure returns either of the following two enumerated values to indicate the Talker Announce status
51 (99.2.4.4) for the given Talker Announce attribute (eTaAttr):

52 1) **Announce Success**: if eTaAttr does not contain a Failure Information sub-TLV (99.4.3.9).

53 2) **Announce Failed**: if eTaAttr contains a Failure Information sub-TLV (99.4.3.9).

54 **99.7.4.8 failTa(eTaAttr, eFailureCode)**

55 This procedure appends the given failure code (eFailureCode) to a Talker Announce attribute (eTaAttr), as follows:

a) Construct a Failure Information sub-TLV using the System ID of this Bridge and the eFailureCode value, in accordance with 99.4.3.9.

b) Append the Failure Information sub-TLV constructed in item a) above to eTaAttr, in accordance with 99.4.3.

c) Return eTaAttr.

99.7.4.9 constructLaAttr(eStreamID, eVID, eLaStatus)

This procedure constructs and returns a Listener Attach attribute (eLaAttr) whose Value field is filled as follows:

```
eLaAttr.StreamID = eStreamID;
eLaAttr.VID = eVID;
eLaAttr.ListenerAttachStatus = eLaStatus;
```

99.7.4.10 constructRaAttr()

This procedure constructs and returns an RA class attribute (eRaAttr) as follows:

a) For each localRaClass entry (eLocalRaClass), construct an RA Class Descriptor sub-TLV (raDescTlv) in accordance with 99.4.2.1 and fill its Value field, as follows:

```
raDescTlv.RaClassID             = eLocalRaClass.id;
raDescTlv.RaClassPriority       = eLocalRaClass.priority;
raDescTlv.RTID                  = eLocalRaClass.rtid;
raDescTlv.RaClassTemplatedDefinedData = eLocalRaClass.templatedDefinedData;
```

b) Construct an RA Class attribute (eRaAttr) and fills its Value field with raDescTlv(s) constructed in item a) above, in accordance with 99.4.2.

c) Return eRaAttr.

99.7.4.11 getLocalRaClass(ePriority)

This procedure searches for a local RA class with the given RA class priority (ePriority), as follows:

```
getLocalRaClass(ePriority) {
  for (eLocalRaClass: localRaClass[*]) {
    if (eLocalRaClass.priority == ePriority) {
      break;
    } else {
      eLocalRaClass = NULL;
    }
  }
  return eLocalRaClass;
}
```

99.7.4.12 setDomainBoundaryStatus(ePortID)

This procedure determines for the given Port (ePortID) the RA class domain boundary status (99.2.2.4) for each local RA class, as follows:

```
setDomainBoundaryStatus(ePortID) {
  for (eLocalRaClass : localRaClass[*]) {
    eNeighborRaClass = neighborRaClass[ePortID, eLocalRaClass.id];
    if (eNeighborRaClass != NULL &&
        eNeighborRaClass.priority == eLocalRaClass.priority)
    {
      portRaClass[ePortID, eLocalRaClass.id].domainBoundaryStatus = FALSE;
    } else {
      portRaClass[ePortID, eLocalRaClass.id].domainBoundaryStatus= TRUE;
    }
  }
}
```

NOTE—The RA class domain boundary status determined by this procedure is used by the Bridge to adjust the operation of priority regeneration, in accordance with 6.9.4.

99.7.4.14 checkResources(eTaDec)

<< Note: It should be considered for D0.6 or subsequent draft to add explicit specification of the operation of the checkResources procedure.>>

This procedure performs resource checking for a Talker Announce declaration contained in the given taDec entry (eTaDec), to determine whether the announced stream is deemed “*Reservable*” or “*Unreservable*”, as follows:

a) eLocalRaClass = **getLocalRaClass**(eTaDec.attr.Priority);

b) Return a Zero value to indicate “*Reservable*”, if all the following conditions are met:

   1) ***Bandwidth***: The bandwidth required by this stream, plus that of all the streams currently reserved in the same RA class, does not exceed the bandwidth budget as indicated by the portRaClass[eTaDec.portID, eLocalRaClass.id].maxBandwidth value [item d) in 99.7.3.8].

   2) ***Latency***: The worst-case maximum latency, computed taking into account all the streams currently reserved in the target RA class plus this stream, does not exceed the delay budget as indicated by the portRaClass[eTaDec.portID, eRaClassID].maxHopDelay value [item e) in 99.7.3.8].

   3) ***Bridge resources***: There are sufficient resources in all the Bridge components (PSFP, queue buffer, shaper, FRER, etc) required by the announced stream.

c) Otherwise, return a RAP failure code (non-zero) associated with the reason why the stream is unreservable.

99.7.4.15 adjustTa(eTaDec)

<< Note: The actual operation of this procedure appears shaper-specific and should be explicitly specified in D0.6 or later.>>

This procedure adjusts a Talker Announce attribute in preparation for a Talker Announce declaration referenced by eTaDec.

99.7.4.16 allocateResources(eTaDec)

<< Note: This procedure implements 99.7.6.2 of D0.5. The operation of this procedure is intended to be specified explicitly in subsequent drafts (D0.6 and later) to ensure interoperability.>>

This procedure allocates resources of the underlying Bridge mechanisms (e.g., traffic shaper) in preparation for the stream referenced by eTaDec.

NOTE – In the normal operation of RAP, allocateResources finally allocates Bridge-internal resources that have been determined to be available by the checkResources procedure [item b)3) in 99.7.4.14] earlier.

99.7.4.17 deallocateResources(eTaDec)

<< Note: This procedure implements 99.7.6.2 of D0.5. The operation of this procedure is intended to be specified explicitly in subsequent drafts (D0.6 and later) to ensure interoperability.>>

This procedure de-allocates resources of the underlying Bridge mechanisms (e.g., traffic shaper) at termination of the stream referenced by eTaDec

NOTE – In the normal operation of RAP, deallocateResources finally de-allocates Bridge-internal resources that have allocated by the allocateResources procedure (99.7.4.17) earlier.

(Annex Z) Collected Issues during Development of this Document

Z.1 Camel-Case vs. Underscore-Case

Several identifiers in P802.1Qdd/D0.5 use an underscore-case notation and should be changed to camel-case notation for consistency, compactness, and readability.

Z.2 VLAN-aware LA attribute

The Listener Attach attribute should be extended by a VID for FRER. The following contents are intended to replace clauses in 99.4.4 of P802.1Qdd/D0.5, followed by a figure to illustrate the issue for discussion.

99.4.4 Listener Attach attribute and TLV encoding

Listener Attach attributes are used in Listener Attach (99.2.5). A Listener Attach attribute encodes in the Value field a set of parameters, as illustrated in Figure 99-19.

Field	Octet	Length
StreamID	1	8
VID	9	1
ListenerAttachStatus	10	1

Figure 99-19—Value of Listener Attach attribute TLV

99.4.4.1 StreamID

An 8-octet field encoding the StreamID element as specified in 46.2.3.1.

99.4.4.2 VID

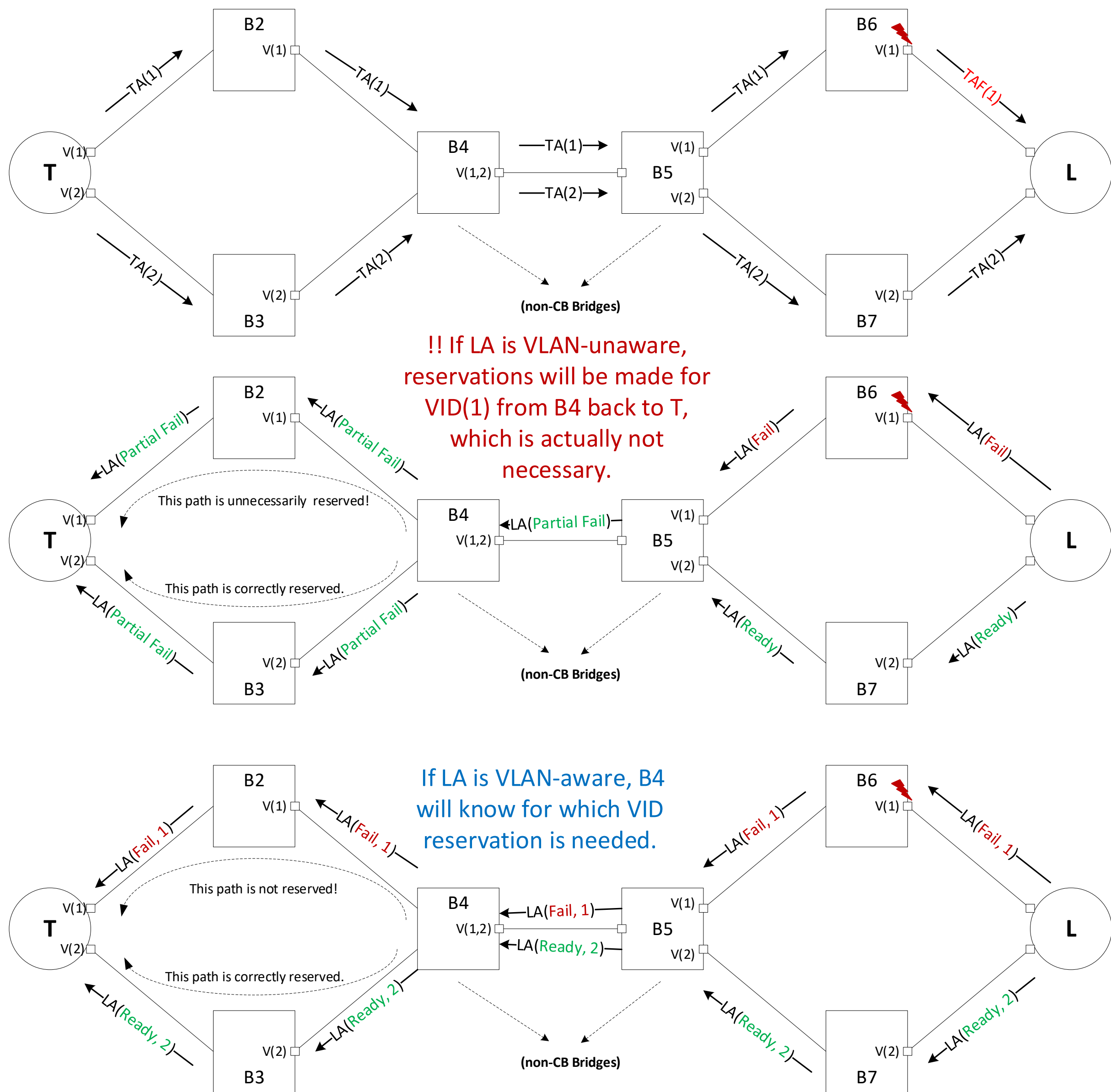
A 1-octet field encoding a VID.

99.4.4.3 ListenerAttachStatus

A 1-octet unsigned integer, taking one of the following three numerical values to indicate the status of the Listener Attach on the Port that declares the Listener Attach attribute:

Table 99.x—Listener Attach status enumeration

Name	Value	Reference
Attach Ready	1	item a) in 99.2.5.2
Attach Partial Fail	2	item b) in 99.2.5.2
Attach Fail	3	item c) in 99.2.5.2



The figure above illustrates what issues may be caused by VLAN-unaware Listener Attach, and how such issues can be resolved by use of VLAN-aware Listener Attach.

Z.4 Sufficient validation of Talker Announce Registrations

RAP Propagator needs to validate each Talker Announce registration for detection and appropriate handling of errors that may be caused by the following reasons:

a) Looping in the network topology (e.g., in the progress of spanning tree reconfiguration).

b) Violation of uniqueness rules in use of Stream ID, Stream DA, etc.

c) Misbehaving Bridge/end stations.

For single-context TA (non-redundant stream), the following error conditions may occur:

- `newTaReg.StreamId == oldTaReg.StreamId &&`
  `newTaReg.portID != oldTaReg.portID`

```
    =>  keep old valid, set new invalid.
```

- `newTaReg.StreamId == oldTaReg.StreamId &&`

```
        newTaReg.portID == oldTaReg.portID &&
       (newTaReg.DA != oldTaReg.DA || newTaReg.VID != oldTaReg.VID ||
        newTaReg.Priority != oldTaReg.Priority )
```

  - ⇨ `set new as "hold" (NOT propagated) until the LA declaration on this port has "LA failed" status or has been withdrawn.`
  - ⇨ `withdraw all existing TA declarations resulting from the old`

Error cases for multiple-context TA (redundant streams) need to be further considered? Also, since LA is VLAN-aware per Z.3, validation of LA registrations seems necessary as well.

Z.5 Issues related to RA class registrations

Issue 1: How a bridge deals with the existing TA registrations on a Port after receiving an updated RA class registration from a neighbor station on that Port?

The current solution is to immediately reprocess all current TA registrations on that Port, based on the updated neighbor RA class information.

To be discussed: should we consider enforcing the neighbor station, e.g. controlled by management, to perform a reset to withdraw all TA declarations before it applies changes to its local RA class configurations.

Issue 2: How to deal with an RA class deregistration event?