EAP Working Group
Internet-Draft
Expires: March 18, 2004

J. Vollbrecht
Vollbrecht Consulting LLC
P. Eronen
Nokia
N. Petroni
University of Maryland
Y. Ohba
TAIS
September 18, 2003

State Machines for EAP Peer and Authenticator
draft-ietf-eap-statemachine-00

Status of this Memo

Copyright Notice

Abstract

This document describes a set of state machines for EAP Peer, EAP Standalone Authenticator (non-passthrough), EAP Backend Authenticator (for use on AAA servers), and EAP Full Authenticator (for both local and

passthrough). This set of state machines shows how EAP can be implemented to support deployment in either a Peer/AP or Peer/AP/AAA Server environment. The Peer and Standalone Authenticator machines are illustrative of how the EAP protocol defined in [I-D.ietf-eap-rfc2284bis] may be implemented. The Backend and Full/Passthrough Authenticators illustrate how EAP/RADIUS protocol support defined in [RFC3579] may be implemented. Where there are differences [I-D.ietf-eap-rfc2284bis]/[RFC3579] are authoritative.

This document describes a state machine based on an EAP "Switch" model. This model includes events and actions for the interaction between the EAP Switch and EAP methods. The State Machine and associated model are informative only. Implementations may achieve the same results using different methods.

A brief description of the EAP "Switch" model is given in the Introduction section.

The authors believe this document corresponds to the current state of revisions to the defining [I-D/ietf-eap-rfc2284bis]/[RFC3579] documents. The intent is for this document to synchronize with the defining documents when they are released, and if discrepancies are found the defining documents are authoritative.

Contents

1 Specification of Requirements

In this document, several words are used to signify the requirements of the specification. These words are often capitalized. The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

2 The EAP Switch Model

This document offers a proposed state machine for RFCs [I-D.ietf-eap-rfc2284bis] and [RFC3579] . There are state machines for the peer, the standalone authenticator, a backend authenticator and a full/passthrough authenticator. Accompanying each state machine diagram is a description of the variables, the functions and the states in the diagram. Whenever possible, the same notation has been used in each of the state machines.

An EAP authentication consists of one or more EAP methods in sequence followed by an EAP Success or EAP Failure sent from the Authenticator to the peer. The EAP Switches control negotiation of EAP methods and sequences of methods.

At both the peer and authenticator one or more EAP methods exist. The EAP switches select which methods each is willing to use, and negotiate between themselves to pick a method or sequence of methods.

```
      Peer             Peer  |  Authenticator       Auth
      Method                 |                      Method
             \               |                    /
               \             |                  /
                Peer         |             Auth
                EAP    <-----|---------->  EAP
                Switch       |             Switch
```

Figure 1: EAP Switch Model

Note that the methods may also have state machines. The details of these are out of scope for this paper.

```
   Peer  |  Authenticator             | Backend
         |               /  Local     |
         |             /    Method    |
   Peer  |        Auth                |        Backend
   EAP --|----->  EAP                 |      -> EAP
  Switch |       Switch               |   /    Server
         |             \              |/
         |               \ passthrough|
         |                            |
```

Figure 2: EAP Passthrough Model

The Full/Passthrough state machine allows a NAS or Edge Device to pass EAP messages to a Backend Server where the real Authentication Method resides. This paper includes a state machine for the EAP authenticator that supports both local and passthrough methods as well as a state machine for the backend authenticator existing at the AAA server. A simple "Standalone" authenticator is also provided to show a basic, non-passthrough authenticator's behavior.

This document describes a set of State Machines that can manage EAP authentication from the peer to an EAP method on the Authenticator or from the Peer through the Authenticator passthrough method to the EAP method on the Backend EAP server.

The state diagrams presented in this document have been coordinated with the diagrams in [IEEE.802-1aa.2003]. The format of the diagrams is adapted from the format therein. Portions of a version of this document are included as Appendix F of [IEEE.802-1aa.2003].

3 Notational conventions used in state diagrams

3.1 Notational specifics

The following state diagrams have been completed based on the conventions specified in [IEEE.802-1aa.2003], section 8.2.1. The complete text is reproduced here:

State diagrams are used to represent the operation of the protocol by a number of cooperating state machines each comprising a group of connected,mutually exclusive states. Only one state of each machine can be active at any given time.

Each state is represented in the state diagram as a rectangular box, divided into two parts by a horizontal line. The upper part contains the state identifier, written in upper case letters. The lower part contains any procedures that are executed on entry to the state.

All permissible transitions between states are represented by arrows, the arrowhead denoting the direction of the possible transition. Labels attached to arrows denote the condition(s) that must be met in order for the transition to take place. All conditions are expressions that evaluate to TRUE or FALSE; if a condition evaluates to TRUE, then the condition is met. The label UCT denotes an unconditional transition (i.e., UCT always evaluates to TRUE). A transition that is global in nature (i.e., a transition that occurs from any of the possible states if the condition attached to the arrow is met) is denoted by an open arrow; i.e., no specific state is identified as the origin of the transition. When the condition associated with a global transition is met, it supersedes all other exit conditions including UCT. The special global condition BEGIN supersedes all other global conditions, and once asserted remains asserted until all state blocks have executed to the point that variable assignments and other consequences of their execution remain unchanged.

On entry to a state, the procedures defined for the state (if any) are executed exactly once, in the order that they appear on the page. Each action is deemed to be atomic; i.e., execution of a procedure completes before the next sequential procedure starts to execute. No procedures execute outside of a state block. The procedures in only one state block execute at a time, even if the conditions for execution of state blocks in different state machines are satisfied, and all procedures in an executing state block complete execution before the transition to and execution of any other state block occurs, i.e., the execution of any state block appears to be atomic with respect to the execution of any other state block and the transition condition to that state from the previous state is TRUE when execution commences. The order of execution of state blocks in different state machines is undefined except as constrained by their transition conditions.A variable that is set to a particular value in a state block retains this value until a subsequent state block executes a procedure that modifies the value.

On completion of all of the procedures within a state, all exit conditions for the state (including all conditions associated with global transitions) are evaluated continuously until one of the conditions is met. The label ELSE denotes a transition that occurs if none of the other conditions for transitions from the state are met (i.e., ELSE evaluates to TRUE if all other possible exit conditions from the state evaluate to FALSE). Where two or more exit conditions with the same level of precedence become TRUE simultaneously, the choice as to which exit condition causes the state transition to take place is arbitrary.

Where it is necessary to split a state machine description across more than one diagram, a transition between two states that appear on different diagrams is represented by an exit arrow drawn with dashed lines, plus a reference to the diagram that contains the destination state. Similarly, dashed arrows and a dashed state box are used on the destination diagram to show the transition to the destination state. In a state machine that has been split in this way, any global transitions that can cause entry to states defined in one of the diagrams are deemed to be potential exit conditions for all of the states of the state machine, regardless of which diagram the state boxes appear in.

Should a conflict exist between the interpretation of a state diagram and either the corresponding global transition tables or the textual description associated with the state machine, the state diagram takes precedence. The interpretation of the special symbols and operators used in the state diagrams is as defined in Section 3.2; these symbols and operators are derived from the notation of the C++ programming language, ISO/IEC 14882. If a boolean variable is described in this clause as being set it has or is assigned the value TRUE, if reset or clear the value FALSE.

In addition to the above notation, there are a couple of clarifications specific to this document. First, all boolean variables are initialized to FALSE before the state machine execution begins. Second, the following notational shorthand is specific to this document:

- <variable> = <expression1> | <expression2> | ...

  Execution of a statement of this form will result in <variable> having a value of exactly one of the expressions. The logic for which of those expressions gets executed is outside of the state machine and could be environmental, configurable, or based on another state machine such as that of the Method.

3.2 State Machine Symbols

- ( )

  Used to force the precedence of operators in Boolean expressions and to delimit the argument(s) of actions within state boxes.

- ;

  Used as a terminating delimiter for actions within state boxes. Where a state box contains multiple actions, the order of execution follows the normal English language conventions for reading text.

- =

  Assignment action. The value of the expression to the right of the operator is assigned to the variable to the left of the operator. Where this operator is used to define multiple assignments, e.g., a = b = X the action causes the value of the expression following the right-most assignment operator to be assigned to all of the variables that appear to the left of the right-most assignment operator.

- !

  Logical NOT operator.

- &&

  Logical AND operator.

- ||

  Logical OR operator.

- if...then...

  Conditional action. If the Boolean expression following the if evaluates to TRUE, then the action following the then is executed.

- { statement 1, ... statement N }

  Compound statement. Braces are used to group statements that are executed together as if they were a single statement.

- !=

  Inequality. Evaluates to TRUE if the expression to the left of the operator is not equal in value to the expression to the right.

- ==

  Equality. Evaluates to TRUE if the expression to the left of the operator is equal in value to the expression to the right.

- <

  Less than. Evaluates to TRUE if the value of the expression to the left of the operator is less than the value of the expression to the right.

- >

  Greater than. Evaluates to TRUE if the value of the expression to the left of the operator is greater than the value of the expression to the right.

- >=

  Greater than or equal to. Evaluates to TRUE if the value of the expression to the left of the operator is either greater than or equal to the value of the expression to the right.

- +

  Arithmetic addition operator.

- -

  Arithmetic subtraction operator.

3.3 Document authority

Should a conflict exist between the interpretation of a state diagram and either the corresponding global transition tables or the textual description associated with the state machine, the state diagram takes precedence. When a discrepancy occurs between any part of this document (text or diagram) and any of the related documents ([I-D.ietf-eap-rfc2284bis], [RFC3579], etc.) the latter (the other document) is considered authoritative and takes precedence.

4 Peer State Machine

The following is a diagram of the EAP Peer state machine. Also included is an explanation of the primitives and procedures referenced in the diagram, as well as a clarification of notation.

(see draft-ietf-eap-statemachine-00.ps for missing diagram if reading [.txt] version)

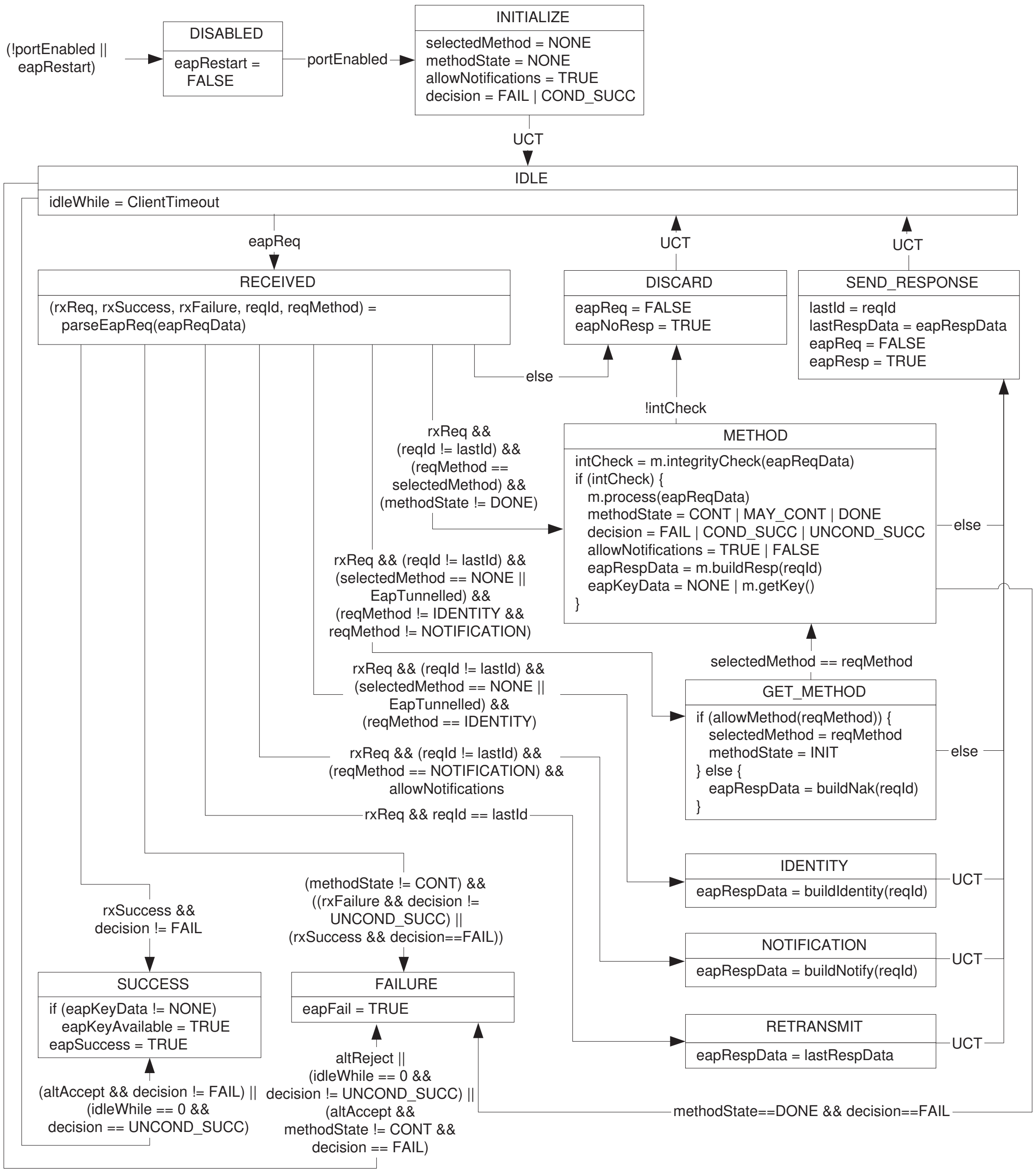

Figure 3: EAP Peer State Machine

4.1 Interface between peer state machine and lower layer

The lower layer presents messages to the EAP peer state machine by storing the packet in eapReqData and setting the eapReq signal to TRUE. Note that despite the name of the signal, the lower layer does not actually inspect the contents of the EAP packet (it could be a Success or Failure message instead of a Request).

When the EAP peer state machine has finished processing the message it sets either eapResp or eapNoResp.

If it sets eapResp, the corresponding response packet is stored in eapRespData. The lower layer is responsible for actually transmitting this message. When the EAP peer state machine authentication is complete it will set eapSuccess or eapFailure to indicate to the lower layer that the authentication has succeeded or failed.

4.1.1 Variables (lower layer to peer)

- eapReq (boolean)

  set to TRUE in lower layer, FALSE in peer state machine. Indicates there is a request available in the lower layer.

- eapReqData (EAP packet)

  set in lower layer when eapReq is set to TRUE. The contents of the available request.

- portEnabled (boolean)

  Indicates that there is a valid port to use for the communication. If at any point the port is not available, portEnabled is set to FALSE and the state machine transitions to DISABLED (or BACKEND_DISABLED).

- idleWhile (integer)

  outside timer used to indicate how long the peer has waited for a new (valid) request.

- altAccept (boolean)

  alternate indication of success, as described in [I-D.ietf-eap-rfc2284bis].

- altReject (boolean)

  alternate indication of failure, as described in [I-D.ietf-eap-rfc2284bis].

4.1.2 Variables (peer to lower layer)

- eapResp (boolean)

  Set to TRUE in peer state machine, FALSE in lower layer. Indicates there is a response to be sent.

- eapNoResp (boolean)

  Set to TRUE in peer state machine, FALSE in lower layer. Indicates the request has been processed, but there is no response to send.

- eapSuccess (boolean)

  Set to TRUE in peer state machine, FALSE in lower layer. Indicates the Peer has reached the SUCCESS state.

- eapFail (boolean)

  Set to TRUE in peer state machine, FALSE in lower layer. Indicates the Peer has reached the FAILURE state.

- eapRespData (EAP Packet)

  Set in peer state machine when eapResp is set to TRUE. The EAP packet which is the response to send.

- eapKeyData (EAP Key)

  Set in peer state machine when keying material becomes available. Set during the METHOD state. Note that this document does not yet define the structure of the type "EAP Key". We expect it to be defined in [I-D.aboba-pppext-key-problem].

- eapKeyAvailable (boolean)

  Set to TRUE in the SUCCESS state if keying material is available. The actual key is stored in eapKeyData.

4.1.3 Constants

- ClientTimeout (integer)

  Configurable amount of time to wait for a valid request before aborting.

- EapTunnelled (boolean)

  Indication of whether EAP is running inside a protected tunnel or not.

4.2 Interface between peer state machine and methods

IN: eapReqData (includes reqId)

OUT: intCheck, eapRespData, allowNotifications, decision

IN/OUT: methodState, (method-specific state)

If methodState==INIT, the method starts by initializing its own method-specific state.

Next, the method must decide whether to process the packet or silently discard it. If the packet looks like it wasn't sent by the legitimate authenticator (for instance, it has invalid MIC, this case should never occur, and the method treats MIC failures as non-fatal), the method can set intCheck=FALSE. In this case, the method should not modify any other variables.

If the method decides to process the packet, it behaves as follows.

- Updates its own method-specific state.
- If the method has derived keying material it wants to export, stores the keying material to eapKeyData.
- Creates a response packet (with the same identifier as the request), and stores it to eapRespData.
- Sets intCheck=TRUE.

Next, the method must update methodState and decision according to the following rules.

- methodState=CONT:

  The method always continues at this point (and the peer wants to continue it). The decision variable is always set to FAIL.

- methodState=MAY_CONT:

  At this point, the authenticator can decide either to continue the method or end the conversation. The decision variable tells us what to do in the case the conversation ends. If the current situation does not satisfy the peer's security policy (that is, if the authenticator now decides to allow access, the peer will not use it), set decision=FAIL. Otherwise, set decision=COND_SUCC.

- methodState=DONE:

  The method always continues at this point, (or the peer sees no point in continuing it).

  If either (a) the authenticator has informed us that it will not allow access, or (b) we're not willing to talk to this authenticator (e.g. our security policy is not satisfied), set decision=FAIL. (Note that this state can occur even if the method still has additional messages left, if continuing it can't change the peer's decision to success).

  If both (a) the server has informed us that it will allow access and the next packet will be EAP Success, and (b) we're willing to use this access, set decision=UNCOND_SUCC.

  Otherwise, we don't know what the server's decision is, but are willing to use the access if the server allows. In this case, set decision=COND_SUCC.

Finally, the method must set the allowNotifications variable. If the new methodState is either CONT or MAY_CONT, and the method specification does not forbid the use of Notification messages, set allowNotifications=TRUE. Otherwise, set allowNotifications=FALSE.

4.3 Peer state machine local variables

4.3.1 Long-term (maintained between packets)

- selectMethod (EAP Type)

  Set in GET_METHOD state. The method the peer believes to be currently "in progress"

- methodState (enumeration)

  As described above.

- lastId (integer)

  Set in SEND_RESPONSE state. The EAP identifier value of the last request.

- lastRespData (EAP packet)

  Set in SEND_RESPONSE state. The EAP packet last sent from the peer.

- decision (enumeration)

  As described above

NOTE: EAP type can be normal type (0..253,255), or an extended type consisting of type 254, Vendor-Id, and Vendor-Type.

4.3.2 Short-term (not maintained between packets)

- rxReq (boolean)

  Set in RECEIVED state. Indicates the current received packet is an EAP request.

- rxSuccess (boolean)

  Set in RECEIVED state. Indicates the current received packet is an EAP Success.

- rxFailure (boolean)

  Set in RECEIVED state. Indicates the current received packet is an EAP Failure.

- reqId (integer)

  Set in RECEIVED state. The identifier value associated with the current EAP request.

- reqMethod (EAP type)

  Set in RECEIVED state. The method type of the current EAP request

- intCheck (boolean)

  Set in METHOD state. Indicates whether the method has decided to accept the current packet.

4.4 Peer state machine procedures

- parseEapReq()

  Determine the code, identifier value, and type of the current request. Also checks that the length field is not longer than the received packet.

- buildNotify()

  Create the appropriate notification response.

- buildIdentity()

  Create the appropriate identity response.

- m.integrityCheck()

  Method-specific procedure to test for the validity of a message.

- m.process()

  Method procedure to parse and process a request for that method.

- m.getKey()

  Method procedure to obtain key material for use by EAP or lower layers.

4.5 Peer state machine states

- DISABLED

  This state is reached anytime service from the lower layer is interrupted or unavailable. Immediate transition to INITIALIZE occurs when the port becomes enabled.

- INITIALIZE

  Initializes variables when the state machine is activated.

- IDLE

  The state machine spends most of its time here, waiting for something to happen.

- RECEIVED

  This state is entered when an EAP packet is received: the packet header is parsed here.

- GET_METHOD

  This state is entered when a request for a new type comes in: either the correct method is started, or a Nak response is built.

- METHOD

  The method processing happens here: the request from the authenticator is processed, and an appropriate response packet is built.

- SEND_RESPONSE

  This state signals the lower layer that a response packet is ready to be sent.

- DISCARD

  This state signals the lower layer that the request was discarded, and no response packet will be sent at this time.

- IDENTITY:

  Handles requests for Identity method, and builds a response.

- NOTIFICATION

  Handles requests for Notification method, and builds a response.

- RETRANSMIT

  Retransmits the previous response packet.

- SUCCESS

  A final state indicating success.

- FAILURE

  A final state indicating failure.

5 Standalone Authenticator State Machine

The following is a diagram of the "Standalone" EAP Authenticator state machine. This diagram should be used for those interested in a self-contained, or non-passthrough, authenticator. Included is an explanation of the primitives and procedures referenced in the diagram, as well as a clarification of notation.

(see draft-ietf-eap-statemachine-00.ps for missing diagram if reading [.txt] version)

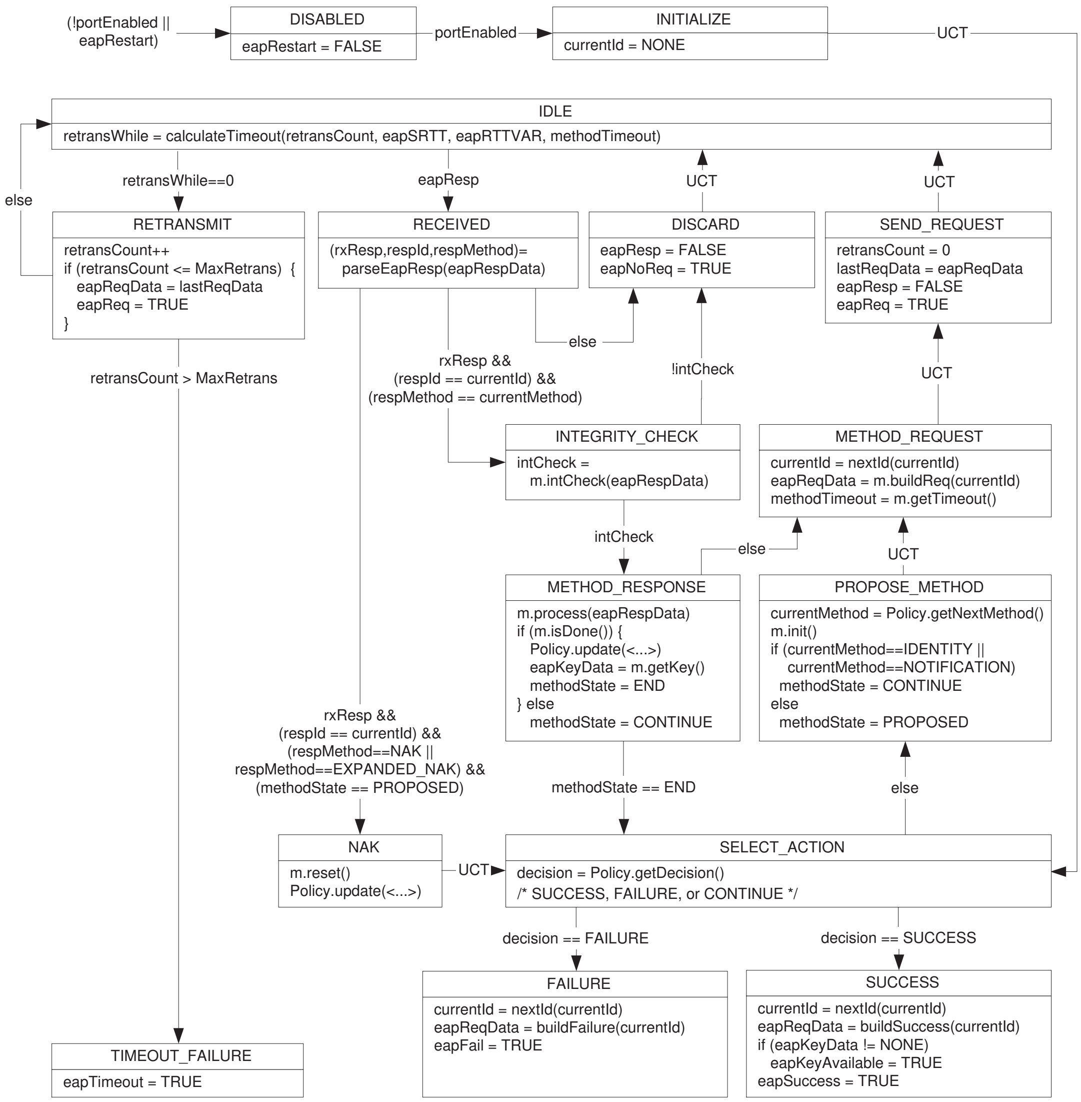

Figure 4: EAP Standalone Authenticator State Machine

5.1 Interface between standalone authenticator state machine and lower layer

The lower layer presents messages to the EAP authenticator state machine by storing the packet in eapResp-Data and setting the eapResp signal to TRUE.

When the EAP authenticator state machine has finished processing the message, it sets one of the signals eapReq, eapNoReq, eapSuccess, and eapFail. If it sets eapReq, eapSuccess, or eapFail, the corresponding request (or success/failure) packet is stored in eapReqData. The lower layer is responsible for actually transmitting this message.

5.1.1 Variables (lower layer to standalone authenticator)

- eapResp (boolean)

  Set to TRUE in lower layer, FALSE in authenticator state machine. Indicates an EAP response is available for processing.

- eapRespData (EAP packet)

  Set in lower layer when eapResp is set to TRUE. The EAP packet to be processed.

- portEnabled (boolean)

  Indicates that there is a valid port to use for the communication. If at any point the port is not available, portEnabled is set to FALSE and the state machine transitions to DISABLED.

- retransWhile (integer)

  Outside timer used to indicate how long the authenticator has waited for a new (valid) response.

- eapRestart (boolean)

  Indicates the lower layer would like to restart authentication

- eapSRTT (integer)

  Smoothed round-trip time. (see [I-D.ietf-eap-rfc2284bis], Section 4.3)

- eapRTTVAR (integer)

  Round-trip time variation. (see [I-D.ietf-eap-rfc2284bis], Section 4.3)

5.1.2 Variables (standalone authenticator to lower layer)

- eapReq (boolean)

  Set to TRUE in authenticator state machine, FALSE in lower layer. Indicates a new EAP request is ready to be sent.

- eapNoReq (boolean)

  Set to TRUE in authenticator state machine, FALSE in lower layer. Indicates the most recent response has been processed, but there is no new request to send.

- eapSuccess (boolean)

  Set to TRUE in authenticator state machine, FALSE in lower layer. Indicates the state machine has reached the SUCCESS state.

- eapFail (boolean)

  Set to TRUE in authenticator state machine, FALSE in lower layer. Indicates the state machine has reached the FAILURE state.

- eapReqData (EAP packet)

  Set in authenticator state machine when eapReq, eapSuccess, or eapFail is set to TRUE. The actual EAP request to be sent (or success/failure).

- eapKeyData (EAP Key)

  Set in authenticator state machine when keying material becomes available. Set during the METHOD state. Note that this document does not yet define the structure of the type "EAP Key". We expect it to be defined in [I-D.aboba-pppext-key-problem].

- eapKeyAvailable (boolean)

  Set to TRUE in the SUCCESS state if keying material is available. The actual key is stored in eapKeyData.

5.1.3 Constants

- MaxRetrans (integer)

  Configurable maximum for how many retransmissions should be attempted before aborting.

5.2 Interface between standalone authenticator state machine and methods

IN: eapRespData, methodState

IN/OUT: currentId, (method-specific state), (policy)

OUT: intCheck, eapReqData

m.init (in: -, out: -)

When the method is first started, it must initialize its own method-specific state, possibly using some information from Policy (e.g. identity).

m.buildReq (in: integer, out: EAP packet)

m.getTimeout (in: -, out: integer or NONE)

Next, the method creates a new EAP Request packet, with the given identifier value, and updates its method-specific state accordingly. The method can also provide a hint for retransmission timeout with m.getTimeout.

m.intCheck (in: EAP packet, out: boolean)

When a new EAP Response is received, the method must first decide whether to process the packet or silently discard it. If the packet looks like it wasn't sent by the legitimate peer (e.g. it has invalid MIC, and this case should never occur), the method can indicate this by returning FALSE. In this case, the method should not modify its own method-specific state.

m.process (in: EAP packet, out: -)

m.isDone (in: -, out: boolean)

m.getKey (in: -, out: EAP key or NONE)

Next, the method processes the EAP Response and updates its own method-specific state. Now the options are to continue the conversation (send another request) or end this method.

If the method wants to end the conversation, it

- Tells Policy about the outcome of the method, and possibly other information.
- If the method has derived keying material it wants to export, returns it from m.getKey().
- Indicates that the method wants to end by returning TRUE from m.isDone().

Otherwise, the method continues by sending another request, as described earlier.

5.3 Standalone authenticator state machine local variables

5.3.1 Long-term (maintained between packets)

- currentMethod (EAP Type)
  EAP type, IDENTITY, or NOTIFICATION.
- currentId (integer)
  0-255 or NONE. Usually updated in PROPOSE_METHOD state. Indicates the identifier value of the currently outstanding EAP request.
- methodState (enumeration)
  As described above.
- retransCount (integer)
  Reset in SEND_REQUEST state and updated in RETRANSMIT state. Current number of retransmissions.
- lastReqData (EAP packet)
  Set in SEND_REQUEST state. EAP packet containing the last sent request.
- methodTimeout (integer)
  Method-provided hint for suitable retransmission timeout, or NONE.

5.3.2 Short-term (not maintained between packets)

- rxResp (boolean)
  Set in RECEIVED state. Indicates the current received packet is an EAP response.
- respId (integer)
  Set in RECEIVED state. The identifier from the current EAP response.
- respMethod (EAP Type)
  Set in RECEIVED state. The method type of the current EAP response.
- intCheck (boolean)
  Set in METHOD state. Indicates whether the method has decided to accept the current packet.
- decision (enumeration)
  Set in SELECT_ACTION state. Temporarily store the policy decision to succeed, fail, or continue.

5.4 EAP standalone authenticator procedures

- calculateTimeout()

  Calculates the retransmission timeout, taking into account the retransmission count, round-trip time measurements, and method-specific timeout hint (see [I-D.ietf-eap-rfc2284bis], Section 4.3).

- parseEapResp()

  Determine the code, identifier value, and type of the current response. Also checks that the length field is not longer than the Received EAP packet

- buildSuccess()

  Create an EAP Success Packet.

- buildFailure()

  Create an EAP Failure Packet.

- nextId()

  Determine the next identifier value to use, based on the previous one.

- Policy.update()

  Update all variables related to internal policy state.

- Policy.getNextMethod()

  Determine the method that should be used at this point in the conversation based on pre-defined policy.

- Policy.getDecision()

  Determine if the policy will allow SUCCESS, FAIL, or is yet to determine (CONTINUE).

- m.intCheck()

  Method-specific procedure to test for the validity of a message.

- m.process()

  Method procedure to parse and process a response for that method.

- m.init()

  Method procedure to initialize state just before use.

- m.reset()

  Method procedure to indicate the method is ending in the middle or before completion.

- m.isDone()

  Method procedure to check for method completion.

- m.getTimeout()

  Method procedure to determine an appropriate timeout hint for that method.

- m.getKey()

  Method procedure to obtain key material for use by EAP or lower layers.

- m.buildReq()

  Method procedure to produce the next request.

5.5 EAP standalone authenticator states

- DISABLED

  The authenticator is disabled until the port is enabled by the lower layer.

- INITIALIZE

  Initializes variables when the state machine is activated.

- IDLE

  The state machine spends most of its time here, waiting for something to happen.

- RECEIVED

  This state is entered when an EAP packet is received: the packet header is parsed here.

- INTEGRITY_CHECK

  A method state in which the integrity of the incoming packet from the peer is verified by the method.

- METHOD_RESPONSE

  A method state in which the incoming packet is processed.

- METHOD_REQUEST

  A method state in which a new request is formulated if necessary.

- PROPOSE_METHOD

  A state in which the authenticator decides which method to try next in the authentication.

- SELECT_ACTION

  In between methods, the state machine re-evaluates whether or not its policy is satisfied and succeeds, fails, or remains undecided.

- SEND_REQUEST

  This state signals the lower layer that a request packet is ready to be sent.

- DISCARD

  This state signals the lower layer that the response was discarded, and no new request packet will be sent at this time.

- NAK

  This state processes Nak responses from the peer.

- RETRANSMIT

  Retransmits the previous request packet.

- SUCCESS

  A final state indicating success.

- FAILURE

  A final state indicating failure.

- TIMEOUT_FAILURE

  A final state indicating failure with no EAP Failure packet sent.

6 EAP Backend Authenticator

When operating in passthrough mode, there are conceptually two parts to the authenticator- the part that passes packets through and the backend that actually implements the EAP method. The following diagram shows a state machine for the backend part of this model when using a AAA server. Note that this diagram is identical to Figure 4 except no retransmit is included in the IDLE state because with RADIUS retransmit is handled by the NAS, and a PICK_UP_METHOD state and variable in INITIALIZE state are added to allow the Method to "pickup" a method started in a NAS. Included is an explanation of the primitives and procedures referenced in the diagram, many of which are the same as above. It should be noted that the "lower layer" in this case is some AAA protocol (e.g. RADIUS).

(see draft-ietf-eap-statemachine-00.ps for missing diagram if reading [.txt] version)

6.1 Interface between backend authenticator state machine and lower layer

The lower layer presents messages to the EAP backend authenticator state machine by storing the packet in aaaEapRespData and setting the aaaEapResp signal to TRUE.

When the EAP backend authenticator state machine has finished processing the message, it sets one of the signals aaaEapReq, aaaEapNoReq, aaaSuccess, and aaaFail. If it sets eapReq, eapSuccess, or eapFail, the corresponding request (or success/failure) packet is stored in aaaEapReqData. The lower layer is responsible for actually transmitting this message.

6.1.1 Variables (AAA interface to backend authenticator)

- aaaEapResp (boolean)

  Set to TRUE in lower layer, FALSE in authenticator state machine. Indicates an EAP response is available for processing.

- aaaEapRespData (EAP packet)

  Set in lower layer when eapResp is set to TRUE. The EAP packet to be processed.

- backendEnabled (boolean)

  Indicates that there is a valid link to use for the communication. If at any point the port is not available, backendEnabled is set to FALSE and the state machine transitions to DISABLED.

6.1.2 Variables (backend authenticator to AAA interface)

- aaaEapReq (boolean)

  Set to TRUE in authenticator state machine, FALSE in lower layer. Indicates a new EAP request is ready to be sent.

- aaaEapNoReq (boolean)

  Set to TRUE in authenticator state machine, FALSE in lower layer. Indicates the most recent response has been processed, but there is no new request to send.

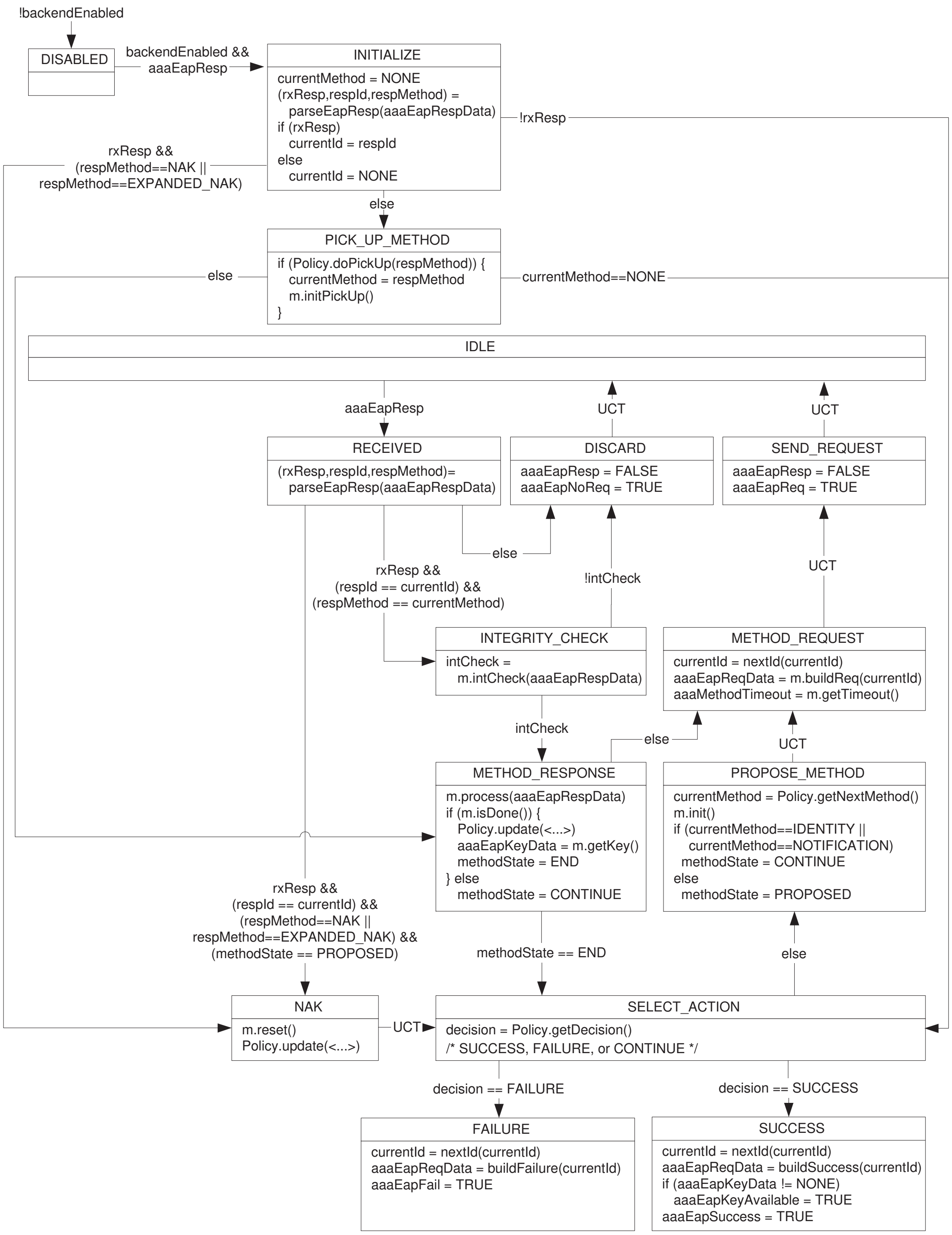

Figure 5: EAP Backend Authenticator State Machine

- aaaSuccess (boolean)

  Set to TRUE in authenticator state machine, FALSE in lower layer. Indicates the state machine has reached the SUCCESS state.

- aaaFail (boolean)

  Set to TRUE in authenticator state machine, FALSE in lower layer. Indicates the state machine has reached the FAILURE state.

- aaaEapReqData (EAP packet)

  Set in authenticator state machine when aaaEapReq, aaaSuccess, or aaaFail is set to TRUE. The actual EAP request to be sent (or success/failure).

- aaaEapKeyData (EAP Key)

  Set in authenticator state machine when keying material becomes available. Set during the METHOD_RESPONSE state. Note that this document does not yet define the structure of the type "EAP Key". We expect it to be defined in [I-D.aboba-pppext-key-problem].

- aaaEapKeyAvailable (boolean)

  Set to TRUE in the SUCCESS state if keying material is available. The actual key is stored in aaaEapKeyData.

- aaaMethodTimeout (integer)

  Method-provided hint for suitable retransmission timeout, or NONE.

6.2 Interface between backend authenticator state machine and methods

The backend method interface is almost the same as in standalone authenticator described in Section 5.2. The only difference is that some methods on the backend may support "picking up" a conversation started by the passthrough. That is, the EAP Request packet was sent by the passthrough, but the backend must process the corresponding EAP Response. Usually only the Identity method supports this, but others are possible.

When "picking up" a conversation, m.initPickUp() is called instead of m.init(). Next, m.process() must examine eapRespData and update its own method-specific state to match what it would have been if it had actually sent the corresponding request. (Obviously, this only works for methods that can determine what the initial request contained; Identity and EAP-TLS are good examples.)

After this, the processing continues as described in Section 5.2

6.3 Backend authenticator state machine local variables

For definitions of the variables used in the Backend Authenticator, see Section 5.3.

6.4 EAP backend authenticator procedures

Most of the procedures of the backend authenticator have already been defined in Section 5.4. This section contains definitions for those not existent in the standalone version, as well as those which are defined differently.

- Policy.doPickUp()

  Notify the policy that an already-chosen method is being picked up and will be completed.
- m.initPickUp()

  Method procedure to initialize state when continuing from an already-started method.

6.5 EAP backend authenticator states

Most of the states of the backend authenticator have already been defined in Section 5.5. This section contains definitions for those not existent in the standalone version, as well as those which are defined differently.

- PICK_UP_METHOD

  Set an initial state for a method that is being continued and was started elsewhere.

7 EAP Full Authenticator

The following two diagrams show the state machine for a complete authenticator. The first diagram is identical to the Standalone State Machine, shown in Figure 4, with the exception that the SELECT_ACTION state has an added transition to PASSTHROUGH. The second diagram also keeps most of the logic except the four method states, and shows how the state machine works once it goes to Passthrough Mode.

The first diagram is largely a reproduction of that found above, with the added hooks for a transition to PASSTHROUGH mode.

(see draft-ietf-eap-statemachine-00.ps for missing diagram if reading [.txt] version)

The second diagram describes the functionality necessary for an authenticator operating in passthrough mode. This section of the diagram is the counterpart of the backend diagram above.

(see draft-ietf-eap-statemachine-00.ps for missing diagram if reading [.txt] version)

7.1 Interface between full authenticator state machine and lower layers

The full authenticator is unique in that it interfaces to multiple lower layers in order to support passthrough mode. The interface to the primary EAP transport layer is the same as described in Section 5. The following describes the interface to the second lower layer, which represents an interface to AAA. It should be noted that there is not necessarily a direct interaction between the EAP layer and the AAA layer, as in the case of [IEEE.802-1aa.2003].

7.1.1 Variables (AAA interface to full authenticator)

- aaaEapReq (boolean)

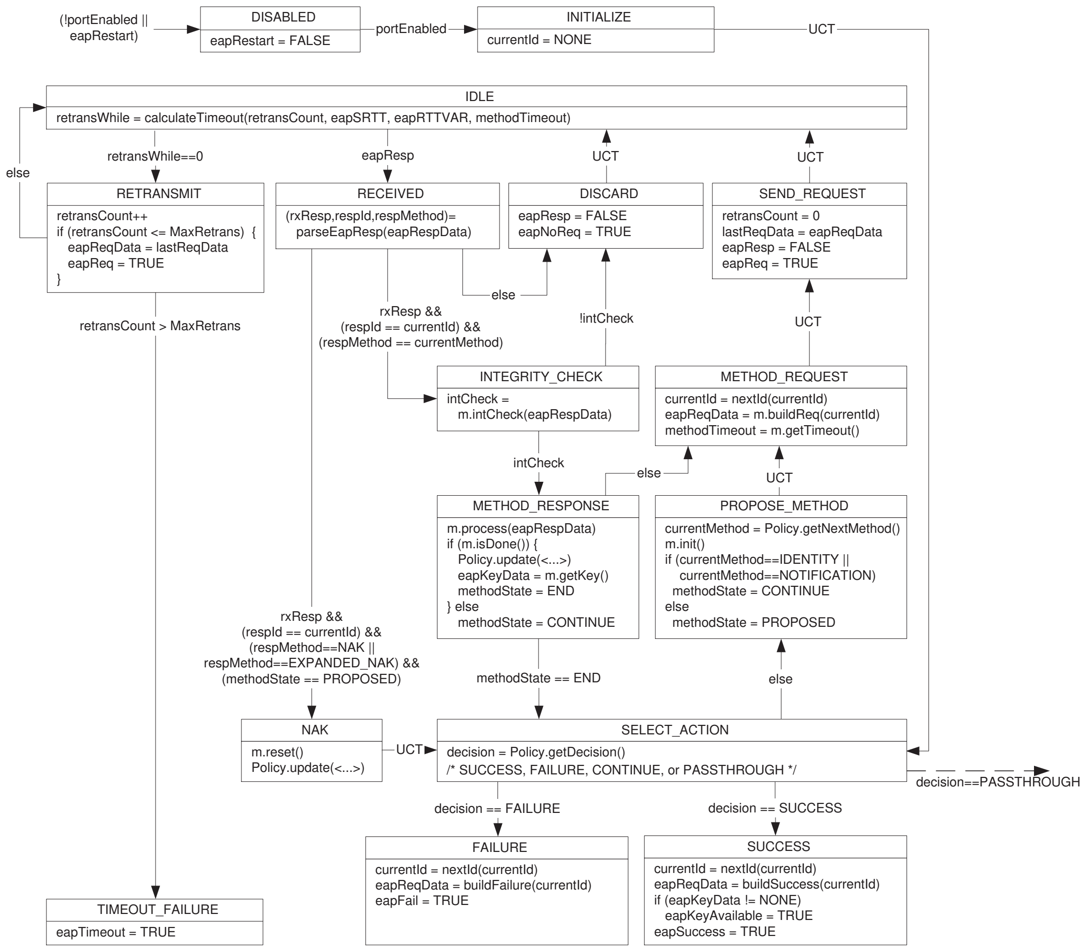

Figure 6: EAP Full Authenticator State Machine (Part 1)

Set to TRUE in lower layer, FALSE in authenticator state machine. Indicates a new EAP request is available from the AAA server.

- aaaEapNoReq (boolean)

  Set to TRUE in lower layer, FALSE in authenticator state machine. Indicates the most recent response has been processed, but there is no new request to send.

- aaaSuccess (boolean)

  Set to TRUE in lower layer. Indicates the AAA backend authenticator has reached the SUCCESS state.

- aaaFail (boolean)

  Set to TRUE in lower layer. Indicates the AAA backend authenticator has reached the FAILURE state.

- aaaEapReqData (EAP packet)

  Set in the lower layer when aaaEapReq, aaaSuccess, or aaaFail is set to TRUE. The actual EAP request to be sent (or success/failure).

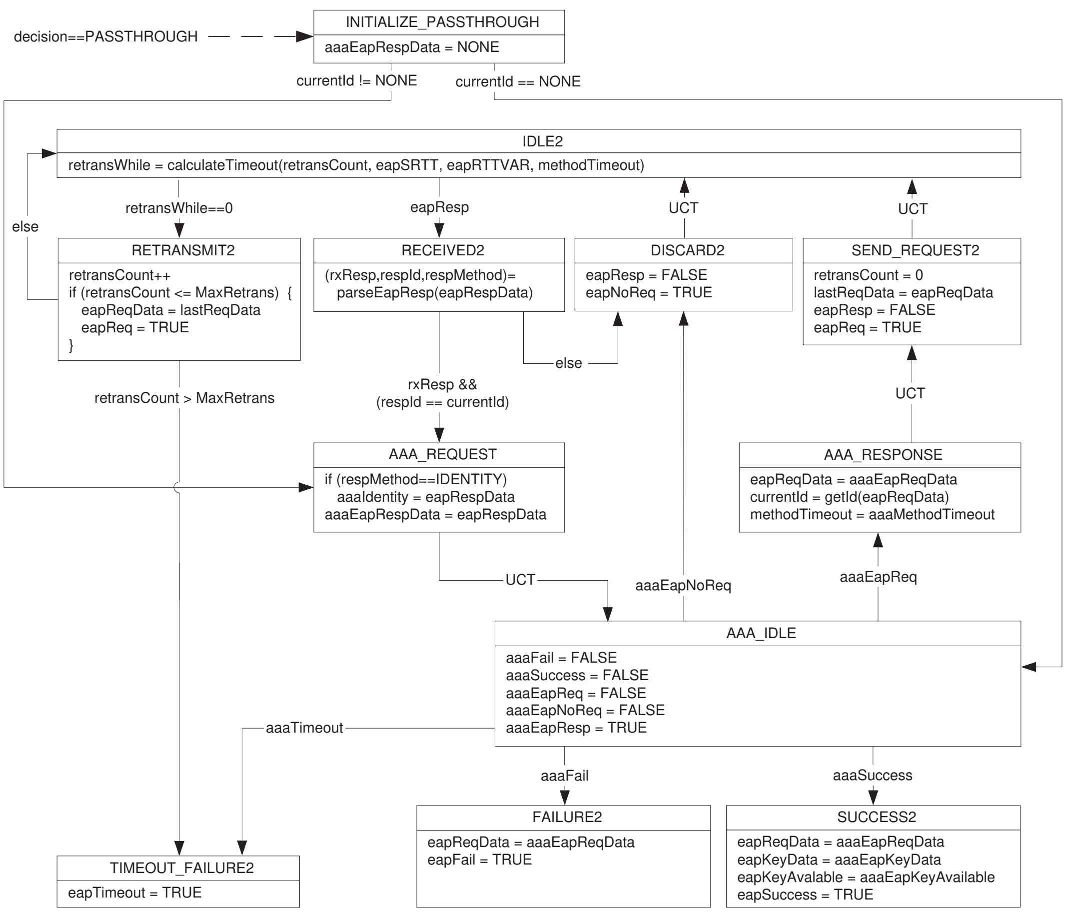

Figure 7: EAP Full Authenticator State Machine (Part 2)

- aaaEapKeyData (EAP Key)

  Set in lower layer when keying material becomes available from the AAA server. Note that this document does not yet define the structure of the type "EAP Key". We expect it to be defined in [I-D.aboba-pppext-key-problem].

- aaaEapKeyAvailable (boolean)

  Set to TRUE in the lower layer if keying material is available. The actual key is stored in aaaEapKeyData.

- aaaMethodTimeout (integer)

  Method-provided hint for suitable retransmission timeout, or NONE.

7.1.2 Variables (full authenticator to AAA interface)

- aaaEapResp (boolean)

  Set to TRUE in authenticator state machine, FALSE in the lower layer. Indicates an EAP response is available for processing by the AAA server.

- aaaEapRespData (EAP packet)

  Set in authenticator state machine when eapResp is set to TRUE. The EAP packet to be processed.

- aaaIdentity (EAP packet)

  Set in authenticator state machine when an IDENTITY response is received. Makes that identity available to AAA lower layer.

- aaaTimeout (boolean)

  Set in AAA_IDLE if after a configurable amount of time there is no response from the AAA layer. The AAA layer in the NAS is itself alive and OK, but for some reason it hasn't received a valid Access-Accept/Reject indication from the backend

7.1.3 Constants

Same as Section 5.

7.2 Interface between full authenticator state machine and methods

Same as standalone authenticator (Section 5.2)

7.3 Full authenticator state machine local variables

Many of the variables of the full authenticator have already been defined in Section 5. This section contains definitions for those not existent in the standalone version, as well as those which are defined differently.

7.3.1 Short-term (not maintained between packets)

- decision (enumeration)

  Set in SELECT_ACTION state. Temporarily store the policy decision to succeed, fail, continue with a local method, or continue in passthrough mode.

7.4 EAP full authenticator procedures

All of the procedures defined in Section 5 exist in the full version. In addition, the following procedures are defined.

- getId()

  Determine the identifier value chosen by the AAA server for the current EAP request.

7.5 EAP full authenticator states

All of the states defined in Section 5 exist in the full version. In addition, the following states are defined.

- INITIALIZE_PASSTHROUGH

  Initializes variables when the passthrough portion of the state machine is activated.

- IDLE2

  The state machine waits for a response from the primary lower layer, which transports EAP traffic from the peer.

- IDLE

  The state machine spends most of its time here, waiting for something to happen.

- RECEIVED2

  This state is entered when an EAP packet is received and the authenticator is in PASSTHROUGH mode: the packet header is parsed here.

- AAA_REQUEST

  The incoming EAP packet is parsed for sending to the AAA server.

- AAA_IDLE

  Idle state which tells the AAA layer it has a response and then waits for a new request, a no-request signal, or success/failure.

- AAA_RESPONSE

  State in which the request from the AAA interface is processed into an EAP request.

- SEND_REQUEST2

  This state signals the lower layer that a request packet is ready to be sent.

- DISCARD2

  This state signals the lower layer that the response was discarded, and no new request packet will be sent at this time.

- RETRANSMIT2

  Retransmits the previous request packet.

- SUCCESS2

  A final state indicating success.

- FAILURE2

  A final state indicating failure.

- TIMEOUT_FAILURE2

  A final state indicating failure with no EAP Failure packet sent.

8 Implementation Considerations

In order to deal with erroneous cases that are not directly related to the protocol behavior, implementations may need additional considerations to provide robustness against errors.

For example, an implementation of a state machine may spend a significant amount of time in a particular state for performing the procedure defined for the state without returning a response. If such an implementation is made on a multithreading system, the procedure may be performed in a separate thread so that the implementation can perform appropriate action to deal with the case without blocking on the state for a long time (or forever if the procedure never completes due to, e.g., a non-responding user or a bug in an application callback function.)

The following states are identified as the possible places of blocking:

- IDENTITY state in the peer state machine. It may take some time to process Identity request when a user input is needed for obtaining an identity from the user. The user may never input an identity. An implementation may define an additional state transition from IDENTITY state to FAILURE state so that authentication can fail if no identity is obtained from the user before ClientTimeout timer expires.
- METHOD state in the peer state machine and in METHOD_RESPONSE state in the authenticator state machines. It may take some time to perform method-specific procedures in these states. An implementation may define an additional state transition from METHOD state and METHOD_RESPONSE state to FAILURE or TIMEOUT_FAILURE state so that authentication can fail if no method processing result is obtained from the method before methodTimeout timer expires.

9 Security Considerations

This document's intent is to describe the EAP state machine fully. To this end, any security concerns with this document are likely a reflection of security concerns with EAP itself.

10 Acknowledgments

The work in this document was done as part of the EAP Design Team. It was done primarily by Nick Petroni, John Vollbrecht, Pasi Eronen and Yoshihiro Ohba. Nick started this work with Bryan Payne and Chuk Seng at the University of Maryland. John Vollbrecht, of Vollbrecht Consulting, started independently with help from Dave Spence at Interlink Networks. John and Nick combined to create a common draft, and then were joined by Pasi Eronen of Nokia who has made major contributions in creating coherent state machines, and Yoshihiro Ohba of Toshiba who insisted on including Passthrough documentation and provided significant support for understanding implementation issues.

In addition significant response and conversation has come from the design team, especially including Jari Arkko of Ericsson and Bernard Aboba of Microsoft as well as the rest of the team. It has also been passed through the 802.1aa group, and has had input from Jim Burns of Meetinghouse and Paul Congdon of Hewlett Packard.

Normative References

[RFC2119]	Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
[RFC2284]	Blunk, L. and J. Vollbrecht, "PPP Extensible Authentication Protocol (EAP)", RFC 2284, March 1998.
[RFC3579]	Aboba, B. and P. Calhoun, "RADIUS (Remote Authentication Dial In User Service) Support For Extensible Authentication Protocol (EAP)", RFC 3579, September 2003.
[I-D.ietf-eap-rfc2284bis]	Blunk, L., "Extensible Authentication Protocol (EAP)", draft-ietf-eap-rfc2284bis-05 (work in progress), September 2003.

Informative References

[I-D.aboba-pppext-key-problem]	Aboba, B. and D. Simon, "EAP Key Management Framework", draft-aboba-pppext-key-problem-07 (work in progress), August 2003.
[IEEE.802-1aa.2003]	Institute of Electrical and Electronics Engineers, "DRAFT Local and Metropolitan Area Networks: Port-Based Network Access Control- Amendment 1", IEEE P802.1aa/D6.1, June 2003.

Authors' Addresses

John R. Vollbrecht
Vollbrecht Consulting LLC
9682 Alice Hill Drive
Dexter, MI 48130
USA
EMail: jrv@umich.edu

Pasi Eronen
Nokia Research Center
P.O. Box 407
FIN-00045 Nokia Group,
Finland
EMail: pasi.eronen@nokia.com

Nick L. Petroni, Jr.
University of Maryland, College Park
A.V. Williams Building
College Park, MD 20742
USA
EMail: npetroni@cs.umd.edu

Yoshihiro Ohba
Toshiba America Information Systems, Inc.
9740 Irvine Blvd.
Irvine, CA 92619-1697
USA
EMail: yohba@tari.toshiba.com

Intellectual Property Statement

The IETF takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Information on the IETF's procedures with respect to rights in standards-track and standards-related documentation can be found in BCP-11. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementors or users of this specification can be obtained from the IETF Secretariat.

The IETF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights which may cover technology that may be required to practice this standard. Please address the information to the IETF Executive Director.

Full Copyright Statement

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.