Internet Engineering Task Force
INTERNET-DRAFT

Vollbrecht et al.

June 30, 2003

State Machines for EAP Peer and Authenticator

Status of this Memo

This Internet-Draft will expire on December 29, 2003.

Copyright Notice

Abstract

This document describes a set of state machines for EAP Peer, EAP Authenticator (supporting local, passthrough and backend), for EAP Passthrough method, and for "backend adapter" that adapts EAP traffic carried by an AAA protocol such as RADIUS or Diameter to a Backend Authenticator. This set of state machines shows how EAP can be implemented to support deployment in either a Peer/AP or Peer/AP/AAA Server environmnet. The Peer and Authenticator machines are illustrative of how the EAP protocol defined in [I-D.ietf-eap-rfc2284bis] may be implemented. The Passtrhough method and "backend adapter" illustrate how EAP protocol support defined in [I-D.aboba-radius-rfc2869bis] may be implemented. Where there are differences [I-D.ietf-eap-rfc2284bis]/ [I-D.aboba-radius-rfc2869bis] 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]/ [I-D.aboba-radius-rfc2869bis] documents. The intent is for this document to synchronize with the defining documents when they are released, and if descrepancies are found the defining documents are authoritative.

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 [I-D.aboba-radius-rfc2869bis] . There are state machines for the peer, the authenticator, a "passthrough method" and a "backend adapter". 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 both the peer and authenticator 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.

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

Figure 1: EAP Switch Model

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

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

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

Figure 2: EAP Switch Model

The Passthrough Method appears to the Authenticator Switch as a method, but its function is to Pass EAP messages to a Backend Server where the real Authentication Method resides. This paper includes a state machine for a Passthrough method and a diagram the flow between an Authenticator with a Passthrough Method and the Backend and its Method.

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 oper ator is either greater than or equal to the value of the expression to the right.

- +

  Arithmetic addition operator.

- -

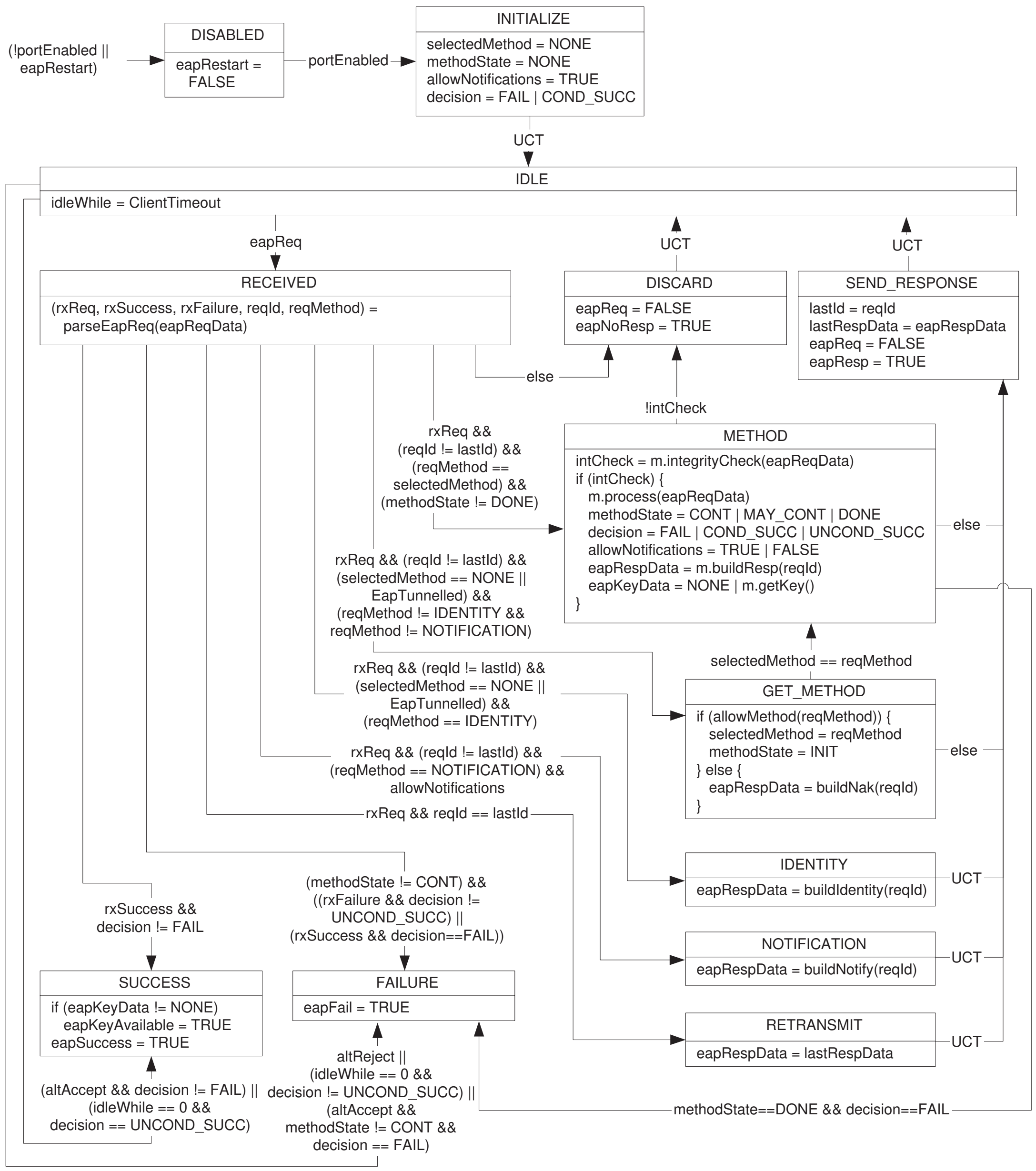
  Arithmetic subtraction operator.

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], [I-D.aboba-radius-rfc2869bis], 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.



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

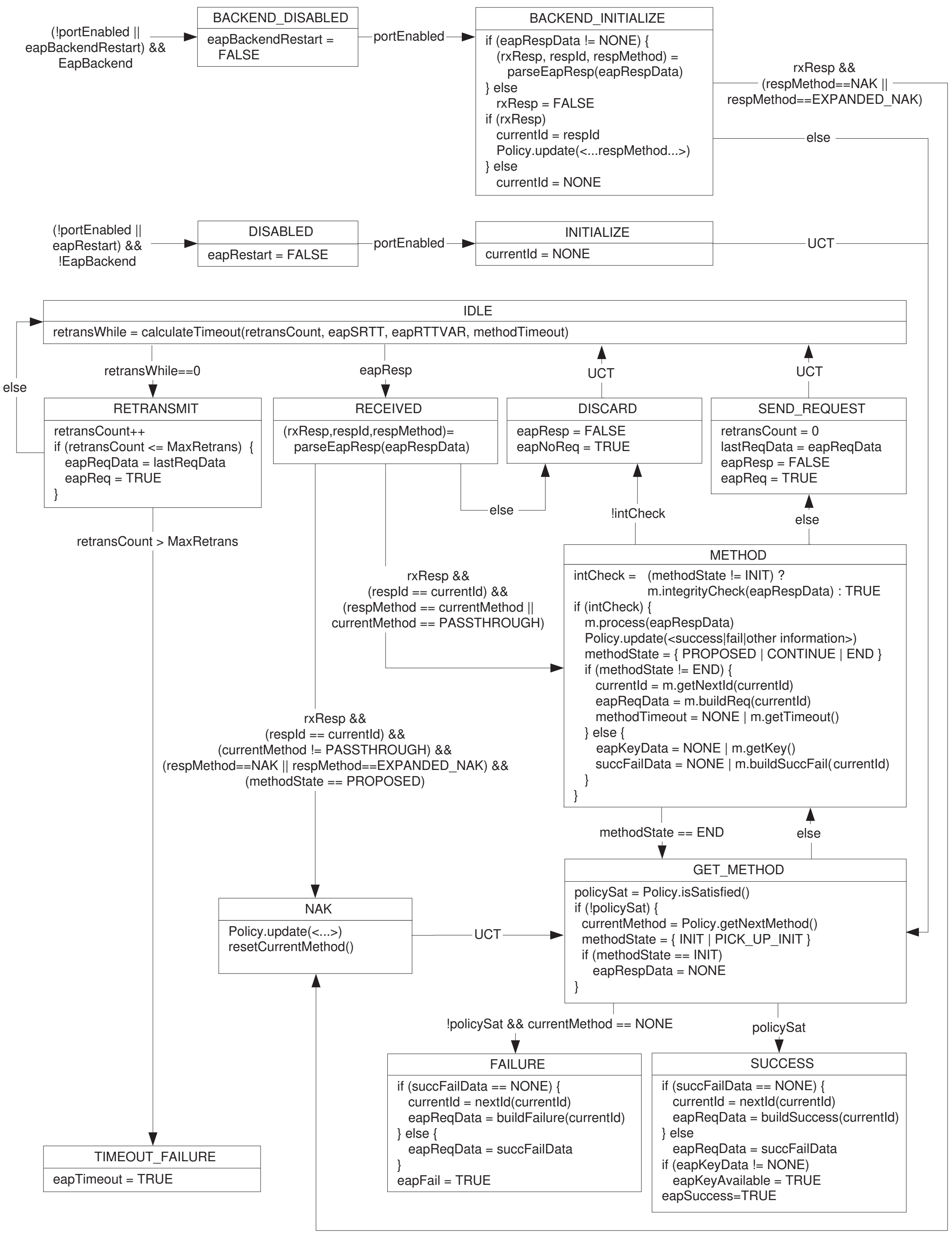
  A final state indicating success.

- FAILURE

  A final state indicating failure.

5 Authenticator State Machine

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


BACKEND_DISABLED
eapBackendRestart = FALSE
(!portEnabled || eapBackendRestart) && EapBackend
portEnabled
BACKEND_INITIALIZE
if (eapRespData != NONE) {
(rxResp, respId, respMethod) = parseEapResp(eapRespData)
} else
rxResp = FALSE
if (rxResp)
currentId = respId
Policy.update(<...respMethod...>)
} else
currentId = NONE
rxResp && (respMethod==NAK || respMethod==EXPANDED_NAK)
else
(!portEnabled || eapRestart) && !EapBackend
DISABLED
eapRestart = FALSE
portEnabled
INITIALIZE
currentId = NONE
UCT
IDLE
retransWhile = calculateTimeout(retransCount, eapSRTT, eapRTTVAR, methodTimeout)
retransWhile==0
eapResp
UCT
UCT
else
RETRANSMIT
retransCount++
if (retransCount <= MaxRetrans) {
eapReqData = lastReqData
eapReq = TRUE
}
RECEIVED
(rxResp,respId,respMethod)= parseEapResp(eapRespData)
DISCARD
eapResp = FALSE
eapNoReq = TRUE
SEND_REQUEST
retransCount = 0
lastReqData = eapReqData
eapResp = FALSE
eapReq = TRUE
else
!intCheck
else
retransCount > MaxRetrans
METHOD
intCheck = (methodState != INIT) ? m.integrityCheck(eapRespData) : TRUE
if (intCheck) {
m.process(eapRespData)
Policy.update(<success|fail|other information>)
methodState = { PROPOSED | CONTINUE | END }
if (methodState != END) {
currentId = m.getNextId(currentId)
eapReqData = m.buildReq(currentId)
methodTimeout = NONE | m.getTimeout()
} else {
eapKeyData = NONE | m.getKey()
succFailData = NONE | m.buildSuccFail(currentId)
}
}
rxResp && (respId == currentId) && (respMethod == currentMethod || currentMethod == PASSTHROUGH)
rxResp && (respId == currentId) && (currentMethod != PASSTHROUGH) && (respMethod==NAK || respMethod==EXPANDED_NAK) && (methodState == PROPOSED)
methodState == END
else
GET_METHOD
policySat = Policy.isSatisfied()
if (!policySat) {
currentMethod = Policy.getNextMethod()
methodState = { INIT | PICK_UP_INIT }
if (methodState == INIT)
eapRespData = NONE
}
NAK
Policy.update(<...>)
resetCurrentMethod()
UCT
!policySat && currentMethod == NONE
policySat
FAILURE
if (succFailData == NONE) {
currentId = nextId(currentId)
eapReqData = buildFailure(currentId)
} else {
eapReqData = succFailData
}
eapFail = TRUE
SUCCESS
if (succFailData == NONE) {
currentId = nextId(currentId)
eapReqData = buildSuccess(currentId)
} else
eapReqData = succFailData
if (eapKeyData != NONE)
eapKeyAvailable = TRUE
eapSuccess=TRUE
TIMEOUT_FAILURE
eapTimeout = TRUE

5.1 Interface between authenticator state machine and lower layer

The lower layer presents messages to the EAP authenticator state machine by storing the packet in eapRespData 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, eapSucess, 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 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 (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

- AuthenticatorTimeout (integer)

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

- MaxRetrans (integer)

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

- EapBackend (boolean)

  Indication of whether the current state machine should be followed as if it is a backend server implementation.

5.2 Interface between authenticator state machine and methods

IN: eapRespData

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

OUT: intCheck, eapReqData, succFailData

A. If methodState==PICK_UP_INIT: This signals that we should "pick up" a conversation that was started by someone else.

Some methods on a backend server may support this feature (usually only Identity, though others are possible). Policy knows what methods support this, and only those methods can end up in PICK_UP_INIT state.

The method behaves as follows.

- Initializes its own method-specific state, possibly using some information from Policy.

- Examines eapRespData, and updates 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.)
- Moves to case C below.

This methodState is also used for the special PASSTHROUGH method, but it is documented elsewhere.

B. If methodState==INIT, we have not sent any requests yet. The method then sends its initial request as follows.

- Initializes its own method-specific state, possibly using some information from Policy (e.g. identity).
- Updates currentId to contain a new identifier value.
- Creates a new request packet (with the new identifier value), and stores it to eapReqData
- If the method is Identity or Notification, sets methodState=CONTINUE; in all other cases, sets methodState=PROPOSED.
- If the method thinks the peer will require more time than usual to respond to this request (for instance, the method most likely requires user input), it can set methodTimeout variable as a hint for suitable retransmission timeout. Otherwise, it sets methodTimeout to NONE.

C. Otherwise we have just received a response.

First 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 peer (e.g. it has invalid MIC, and this case should never occur), the method can set intCheck=FALSE. In this case, the method must not modify methodState or currentId, and it should not modify its own method-specific state.

If the packet is accepted, the options are to continue the conversation (send another request) or end the conversation. If the conversation is continued, the method behaves as follows.

- Updates its own method-specific state.
- Possibly tells Policy something.
- Updates currentId to contain a new identifier value.
- Creates a new request packet (with the new identifier value), and stores it to eapReqData.
- Sets methodState=CONTINUE and intCheck=TRUE.
- Sets methodTimeout as described above.

If the method wants to end the conversation,

- Tells Policy about the outcome of the method, and possibly other information.

- If the method has derived keying material it wants to export, stores the keying material to eapKeyData.
- Sets succFailData=NONE (except special PASSTHROUGH method).
- Sets methodState=END.

5.3 Authenticator state machine local variables

5.3.1 Long-term (maintained between packets)

- currentMethod (EAP Type)

  EAP type, PASSTHROUGH, or NONE.
- currentId (integer)

  0-255 or NONE. Usually updated in METHOD state. Indicates the identifier value of the currently outstanding EAP request.
- methodState (enumeration)

  As described above.
- retransCount (integer)

  Set in SEND_REQUEST 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.
- succFailData (EAP packet)

  Set in METHOD state. Usage described above.
- intCheck (boolean)

  Set in METHOD state. Indicates whether the method has decided to accept the current packet.
- policySat (boolean)

  Set in GET_METHOD state. Stored value of last call to Policy.isSatisfied().

5.4 EAP authenticator procedures

- 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.

- resetCurrentMethod()

  Alert the current method that it has been aborted.

- 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.isSatisfied()

  Determine if the policy will allow SUCCESS or not.

- m.integrityCheck()

  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.buildSuccFail()

  Only used for the special passthrough method. Constructs a Success or Failure packet based on the EAP payload contained in the AAA accept or reject message.

- m.buildReq()

  Method procedure to produce the next request.

- m.getNextId()

  Method procedure that is the parallel of the switch-level nextId(). Often it is up to the method to decide the next ID (particularly in backend authenticators).

- m.getKey()

  Method procedure to obtain key material for use by EAP or lower layers. Also checks that the length field is not longer than the received EAP packet.

5.5 EAP authenticator states

- DISABLED

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

- BACKEND_DISABLED

  Same for backend server.

- INITIALIZE

  Initializes variables when the state machine is activated.

- BACKEND_INITIALIZE

  Same for backend server. Also parses the headers of initial response packet (a response to a request sent by the NAS), if any.

- GET_METHOD

  This state chooses what should happen next: either a method is started, or the conversation is ended.

- 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.

- METHOD

  This state builds request packets and processes responses received from the peer. For passthrough mode, see Section 6.3.

- 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 Passthrough and backend

6.1 Overview

There are two different cases to consider: either the first EAP request is sent by the NAS, or by the backend.

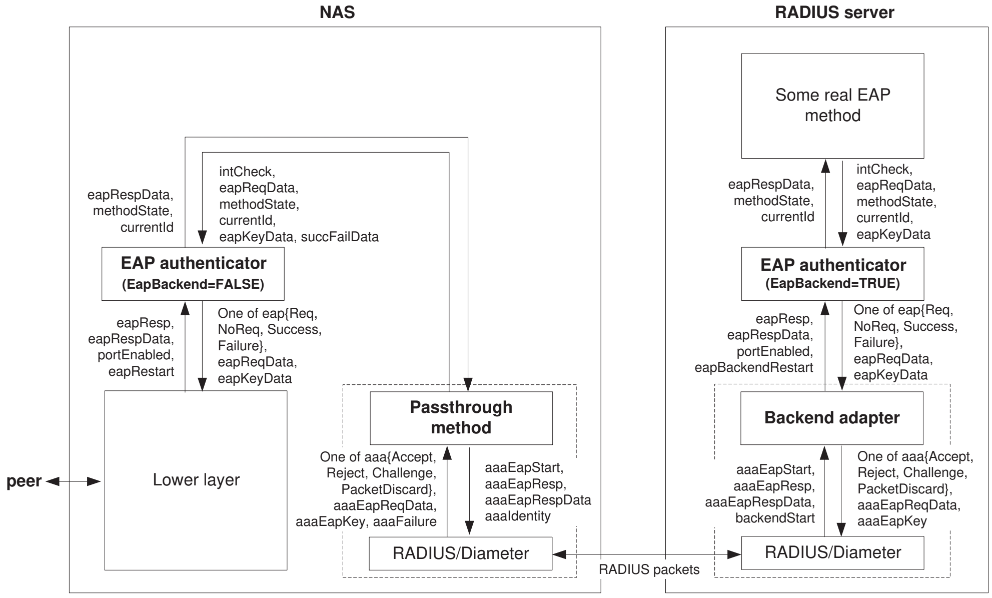
In the simple case, the NAS signals the backend that it wants to start an EAP conversation, and the backend sends the first EAP Request. This case could be handled with a very simple passthrough state machine.

The complex case is where the first EAP request (or several requests) are sent by the NAS, and at some point, the NAS switches to passthrough mode and backend takes over. This requires very careful handling to get right! After the backend has sent its first request, the situation is normal.

6.2 State machine overview

We could have drawn a separate passthrough authenticator and backend authenticator state diagrams, but this was not done because a) this way we can describe an authenticator that starts, b) the differences between the diagrams would have been rather small

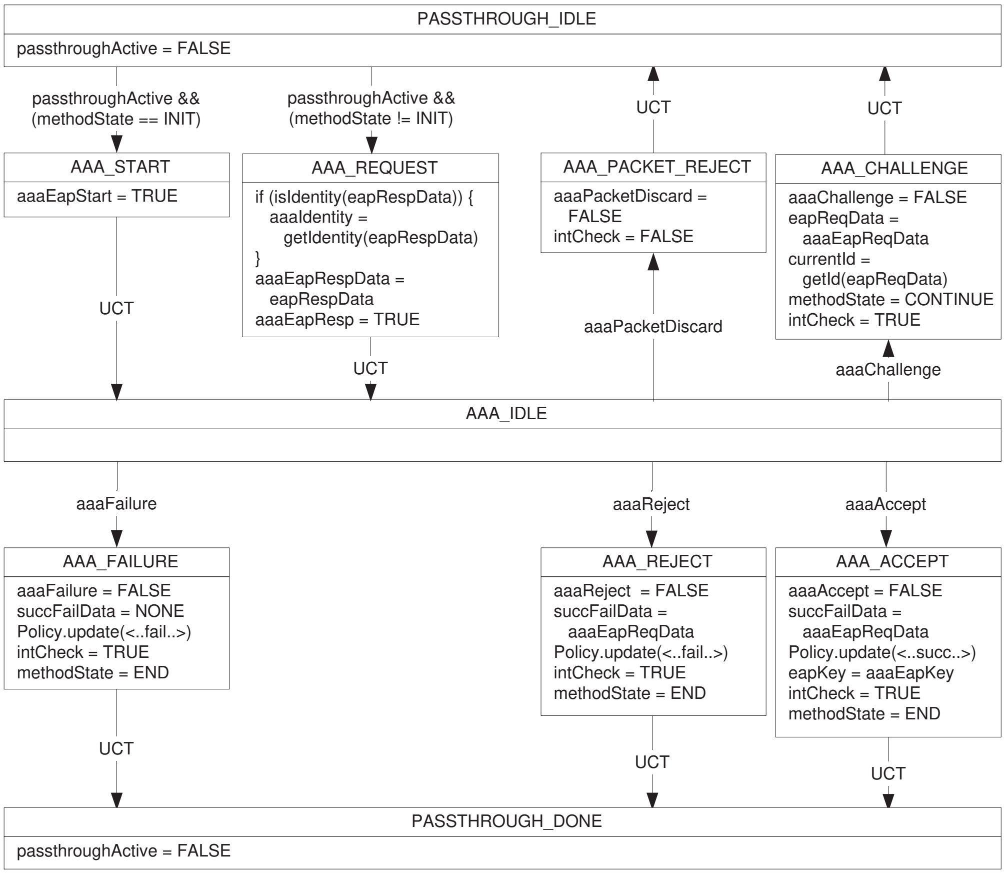
The passthrough/backend split is specified using two copies of the authenticator state machine, one in the "passthrough" processor, and one in the "backend" processor.



The passthrough method takes EAP responses, encapsulates them into RADIUS Access-Request packets, and sends them to the backend. The passthrough method also translates the RADIUS response packet to appropriate EAP requests and signals to EAP authenticator switch.

The backend adapter state machine receives EAP responses from NAS inside RADIUS Access-Request packets, and passes these to the authenticator state machine using the normal lower layer interface. It also relays the signals returned by the authenticator state machine (e.g. eapReq, eapSuccess) to NAS using appropriate RADIUS messages.

6.3 Passthrough



If the first EAP request is sent by the backend, the NAS Policy simply activates the PASSTHROUGH method with methodState==INIT. The PASSTHROUGH method sends a RADIUS packet signalling EAP-Start to the backend, and relays messages.

In the case where the first EAP request is originated from the NAS, the NAS behaves as follows: Policy starts the (local) Identity method, which sends a request packet. When the response comes back, the Identity method processes it and sets methodState=END.

Policy then starts the PASSTHROUGH method with methodState PICK_UP_INIT. The PASSTHROUGH method sends the Identity response to the backend server (inside a RADIUS Access-Request packet). The backend server responds with some other request (passed PASSTHROUGH method inside a RADIUS Access-Challenge), and PASSTHROUGH method relays this request to the peer.

This continues until the backend server responds with Access-Accept or Accept-Reject. The PASSTHROUGH method notifies the Policy, stores the Success/Failure packet to succFailData, and sets methodState=END.

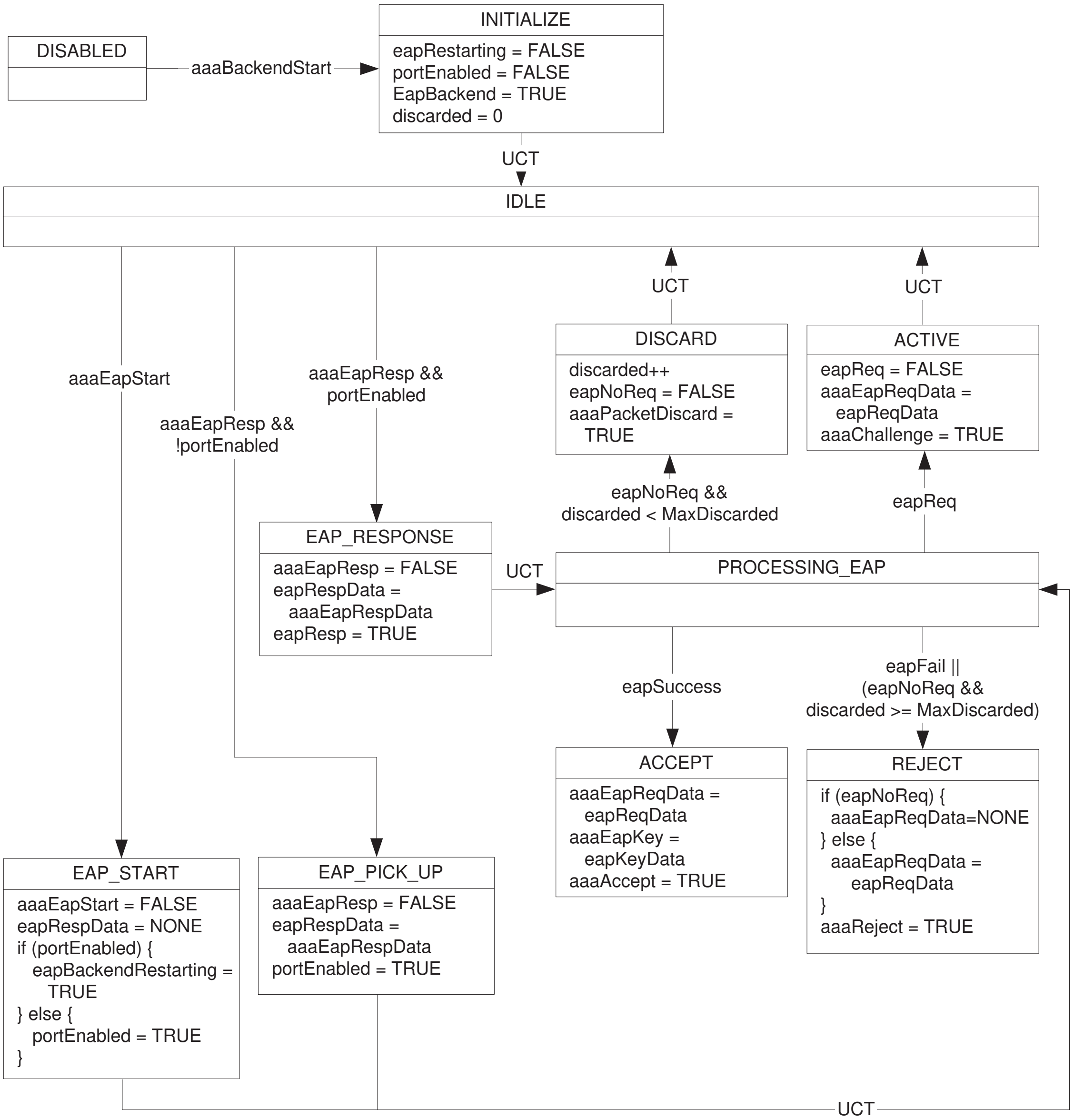
The policy then moves to SUCCESS or FAILURE state, and the Success/ Failure packet is sent to the peer.

NOTE: When using the PASSTHROUGH method, the METHOD state in the Authenticator diagram is completely replaced with the lines

```
passthroughActive=TRUE

while (passthroughActive) {}
```

6.4 Backend



The case where the conversation is started by the backend server (passthrough signals EAP-Start) does not require any special handling. The only difference to ordinary authenticator is that EAP retransmissions are handled by the NAS (AuthenticatorTimeout is set to INFINITY).

The case where the conversation is started by NAS requires special handling though. The client's response is passed to the backend in the first Access-Request.

The backend must process this request BEFORE it sends anything, for the following reasons:

- The identifier chosen for the next packet must be different from the one contained in client's response.

- The client's response will probably contain some useful information (e.g. Identity response).

If the passthrough wants the backend server to send the first EAP Request, it sets aaaEapStart=TRUE. Otherwise, when the passthrough receives an EAP Response from the peer, it stores it in aaaEapRespData, and sets aaaEapResp=TRUE.

These signals are sent to the backend using RADIUS or DIAMETER packets. In RADIUS, aaaEapStart signal corresponds to an Access-Request signifying EAP-Start (EAP-Message attribute with no data), and aaaEapResp corresponds to an ordinary Access-Request (with aaaEapRespData stored in EAP-Message attribute). See [I-D.aboba-radius-rfc2869bis] for details.

When the backend has finished processing the signal, it sets one of the variables aaaAccept, aaaReject, aaaChallenge, aaaPacketDiscard, and unless aaaPacketDiscard is set, also aaaEapReqData. If aaaAccept is set, aaaEapKey can also be set.

These signals are returned to the passthrough using RADIUS or DIAMETER packets. In RADIUS, the first three signals correspond to Access-Accept, Access-Reject, and Access-Challenge packets, with aaaEapReqData stored in EAP-Message attribute. The fourth signal (aaaPacketDiscard) corresponds to Access-Challenge with Error-Cause set to 202. See [I-D.aboba-radius-rfc2869bis] for details.

If the passthrough does not receive a valid RADIUS response, even after retransmissions (handled by RADIUS), it sets aaaFailure. The passthrough can also set variable aaaIdentity, which contains the last Identity response seen. This can be used to route the RADIUS message to correct destination.

6.5 Variables (passthrough forwarder to backend adapter, via AAA protocol)

- aaaEapStart (boolean)

  Set to TRUE in the passthrough forwarder; signals the backend adapter that it should send the first EAP Request. (Transported as a RADIUS Access-Request with an empty EAP-Message attribute)

- aaaEapResp (boolean)

  Set to TRUE in the passthrough forwader; signals the backend adapter that an EAP Response is available (Transported as an RADIUS Access-Request containing EAP-Message attribute containing aaaEapRespData).

- aaaEapRespData (EAP packet)

  See above.

6.6 Variables (backend adapter to passthrough forwarder, via AAA protocol)

- aaaAccept (boolean)

  (Transported as RADIUS Access-Accept)

- aaaReject (boolean)

  (Transported as RADIUS Access-Reject)

- aaaChallenge (boolean)

  (Transported as RADIUS Access-Challenge)

- aaaPacketDiscard (boolean)

  (Transported as RADIUS Access-Challenge with Error-Cause attribute set to 202)

- aaaEapReqData (EAP packet or NONE)

  Set when aaaChallenge, aaaAccept, or aaaReject is set to TRUE. Contains the actual EAP packet to be sent (Transported as EAP-Message attribute).

- aaaEapKey (EAP Key)

  May be set when aaaAccept is set to true. (Usually transported as vendor-specific MS-MPPE-Send-Key/MS-MPPE-Recv-Key attributes).

6.7 Variables (passthrough forwarder to AAA protocol)

- aaaIdentity (octet string)

  Contains the identity from the last received EAP Identity Response.

6.8 Variables (AAA protocol to passthrough forwarder)

- aaaFailure (boolean)

  Set to TRUE by AAA protocol if it fails to receive a valid answer from the AAA server.

6.9 Variables (AAA protocol to backend adapter)

- aaaBackendStart (boolean)

  MaxDiscarded (integer constant)

6.10 Variables (local to backend adapter)

- discarded (integer)

  Counts the number of packets that have been silently discarded.

6.11 Procedures

- isIdentity()

  Determines if the EAP packet given is argument contains an Identity response.

- getIdentity()

  Returns the identity contain in a Identity response packet.

7 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.

8 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.

[I-D.ietf-eap-rfc2284bis]
Blunk, L., "Extensible Authentication Protocol (EAP)", draft-ietf-eap-rfc2284bis-04 (work in progress), June 2003.

[I-D.aboba-radius-rfc2869bis]
Aboba, B. and P. Calhoun, "RADIUS Support For Extensible Authentication Protocol (EAP)", draft-aboba-radius-rfc2869bis-22 (work in progress), May 2003.

Informative References

[I-D.aboba-pppext-key-problem]

Aboba, B. and D. Simon, "EAP Keying Framework", draft-aboba-pppext-key-problem-06 (work in progress), March 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
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.
Department of Computer Science
University of Maryland
A.V. Williams Building
College Park, MD 20742
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

Expiration Date

This memo is filed as <draft-vollbrecth-eap-state-04.ps>, and expires December 29, 2003.