Network Working Group
Request for Comments: 4137
Category: Informational

J. Vollbrecht
Meetinghouse Data Communications
P. Eronen
Nokia
N. Petroni
University of Maryland
Y. Ohba
TARI

August 2005

State Machines for Extensible Authentication Protocol (EAP) Peer and Authenticator

Status of This Memo

Copyright Notice

Abstract

This document describes a set of state machines for Extensible Authentication Protocol (EAP) peer, EAP stand-alone authenticator (non-pass-through), EAP backend authenticator (for use on Authentication, Authorization, and Accounting (AAA) servers), and EAP full authenticator (for both local and pass-through). This set of state machines shows how EAP can be implemented to support deployment in either a peer/authenticator or peer/authenticator/AAA Server environment. The peer and stand-alone authenticator machines are illustrative of how the EAP protocol defined in RFC 3748 may be implemented. The backend and full/pass-through authenticators illustrate how EAP/AAA protocol support defined in RFC 3579 may be implemented. Where there are differences, RFC 3748 and RFC 3579 are authoritative.

The state machines are based on the EAP "Switch" model. This model includes events and actions for the interaction between the EAP Switch and EAP methods. A brief description of the EAP "Switch" model is given in the Introduction section.

The state machine and associated model are informative only. Implementations may achieve the same results using different methods.

Table of Contents

1. Introduction: The EAP Switch Model

This document offers a proposed state machine for RFCs [RFC3748] and [RFC3579]. There are state machines for the peer, the stand-alone authenticator, a backend authenticator, and a full/pass-through 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.

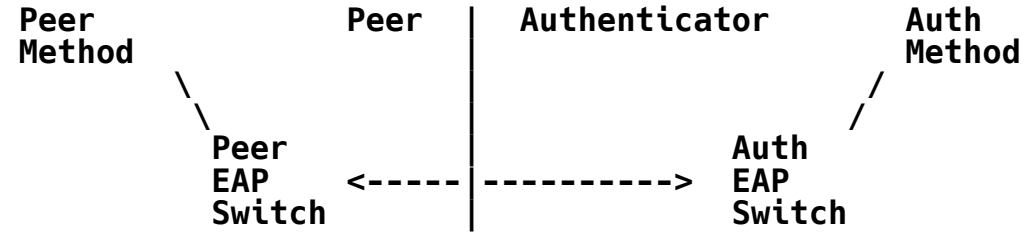


Figure 1: EAP Switch Model

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.

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

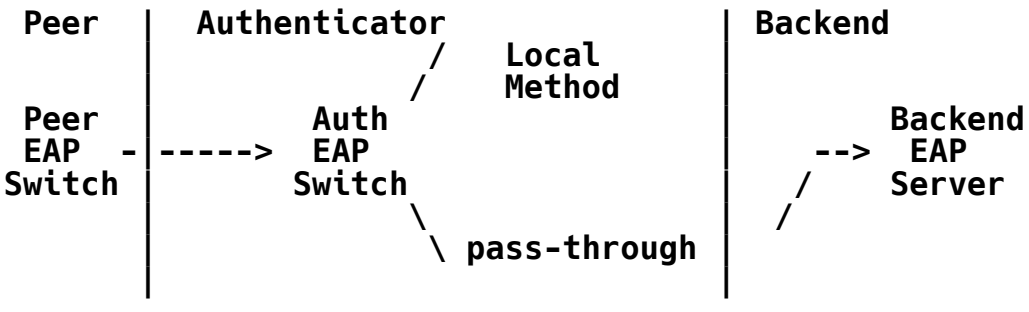


Figure 2: EAP Pass-Through Model

The Full/Pass-Through state machine allows an NAS or edge device to pass EAP Response messages to a backend server where the authentication method resides. This paper includes a state machine for the EAP authenticator that supports both local and pass-through methods as well as a state machine for the backend authenticator existing at the AAA server. A simple stand-alone authenticator is also provided to show a basic, non-pass-through 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 pass-through method to the EAP method on the backend EAP server.

Some environments where EAP is used, such as PPP, may support peer-to-peer operation. That is, both parties act as peers and authenticators at the same time, in two simultaneous and independent EAP conversations. In this case, the implementation at each node has to perform demultiplexing of incoming EAP packets. EAP packets with code set to Response are delivered to the authenticator state machine, and EAP packets with code set to Request, Success, or Failure are delivered to the peer state machine.

The state diagrams presented in this document have been coordinated with the diagrams in [1X-2004]. The format of the diagrams is adapted from the format therein. The interface between the state machines defined here and the IEEE 802.1X-2004 state machines is also explained in Appendix F of [1X-2004].

2. 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" are to be interpreted as described in [RFC2119].

3. Notational Conventions Used in State Diagrams

3.1. Notational Specifics

The following state diagrams have been completed based on the conventions specified in [1X-2004], 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 uppercase letters. The lower part contains any procedures that are executed upon 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 it 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 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. That is, 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 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 potential exit conditions for all 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 it is described as being reset or clear, it has 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. If 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. If 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 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.

>

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.

<=

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

++

Increment the preceding integer operator by 1.

+

Arithmetic addition operator.

&

Bitwise AND 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 ([RFC3748], [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 the .pdf version for missing diagram or refer to Appendix A.1 if reading the .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 that a request is 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 the EAP peer state machine should be ready for communication. This is set to TRUE when the EAP conversation is started by the lower layer. If at any point the communication port or session is not available, portEnabled is set to FALSE, and the state machine transitions to DISABLED. To avoid unnecessary resets, the lower layer may dampen link down indications when it believes that the link is only temporarily down and that it will

soon be back up (see [RFC3748], Section 7.12). In this case, portEnabled may not always be equal to the "link up" flag of the lower layer.

idleWhile (integer)

Outside timer used to indicate how much time remains before the peer will time out while waiting for a valid request.

eapRestart (boolean)

Indicates that the lower layer would like to restart authentication.

altAccept (boolean)

Alternate indication of success, as described in [RFC3748].

altReject (boolean)

Alternate indication of failure, as described in [RFC3748].

4.1.2. Variables (peer to lower layer)

eapResp (boolean)

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

eapNoResp (boolean)

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

eapSuccess (boolean)

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

eapFail (boolean)

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

eapRespData (EAP packet)

Set in peer state machine when eapResp is set to TRUE. The EAP packet that 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 define the structure of the type "EAP key". We expect that it will be defined in [Keying].

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, initialized by implementation-specific means (e.g., a configuration setting).

4.2. Interface between Peer State Machine and Methods

IN: eapReqData (includes reqId)

OUT: ignore, eapRespData, allowNotifications, decision

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

The following describes the interaction between the state machine and EAP methods.

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

Next, the method must decide whether to process the packet or to discard it silently. If the packet appears to have been sent by someone other than the legitimate authenticator (for instance, if message integrity check fails) and the method is capable of treating such situations as non-fatal, the method can set ignore=TRUE. In this case, the method should not modify any other variables.

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

- It updates its own method-specific state.
- If the method has derived keying material it wants to export, it stores the keying material to eapKeyData.
- It creates a response packet (with the same identifier as the request) and stores it to eapRespData.
- It sets ignore=FALSE.

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 to end the conversation. The decision variable tells us what to do if 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 never 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 cannot 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 do not 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 if 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 that the peer believes is currently "in progress"

methodState (enumeration)

As described above.

lastId (integer)

0-255 or NONE. 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 that the current received packet is an EAP request.

rxSuccess (boolean)

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

rxFailure (boolean)

Set in RECEIVED state. Indicates that 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.

ignore (boolean)

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

4.4. Peer State Machine Procedures

NOTE: For method procedures, the method uses its internal state in addition to the information provided by the EAP layer. The only arguments that are explicitly shown as inputs to the procedures are those provided to the method by EAP. Those inputs provided by the method's internal state remain implicit.

parseEapReq()

Determine the code, identifier value, and type of the current request. In the case of a parsing error (e.g., the length field is longer than the received packet), rxReq, rxSuccess, and rxFailure will all be set to FALSE. The values of reqId and reqMethod may be undefined as a result. Returns three booleans, one integer, and one EAP type.

processNotify()

Process the contents of Notification Request (for instance, display it to the user or log it). The return value is undefined.

buildNotify()

Create the appropriate notification response. Returns an EAP packet.

processIdentity()

Process the contents of Identity Request. Return value is undefined.

buildIdentity()

Create the appropriate identity response. Returns an EAP packet.

m.check()

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

m.process()

Method procedure to parse and process a request for that method. Returns a methodState enumeration, a decision enumeration, and a boolean.

m.buildResp()

Method procedure to create a response message. Returns an EAP packet.

m.getKey()

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

4.5. Peer State Machine States

DISABLED

This state is reached whenever 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. Stand-Alone Authenticator State Machine

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

(see the .pdf version for missing diagram or refer to Appendix A.2 if reading the .txt version)

Figure 4: EAP Stand-Alone Authenticator State Machine

5.1. Interface between Stand-Alone 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, 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 Stand-Alone Authenticator)

eapResp (boolean)

Set to TRUE in lower layer, FALSE in authenticator state machine. Indicates that 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 the EAP authenticator state machine should be ready for communication. This is set to TRUE when the EAP conversation is started by the lower layer. If at any point the communication port or session is not available, portEnabled is set to FALSE, and the state machine transitions to DISABLED. To avoid unnecessary resets, the lower layer may dampen link down indications when it believes that the link is only temporarily down and that it will

soon be back up (see [RFC3748], Section 7.12). In this case, portEnabled may not always be equal to the "link up" flag of the lower layer.

retransWhile (integer)

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

eapRestart (boolean)

Indicates that the lower layer would like to restart authentication.

eapSRTT (integer)

Smoothed round-trip time. (See [RFC3748], Section 4.3.)

eapRTTVAR (integer)

Round-trip time variation. (See [RFC3748], Section 4.3.)

5.1.2. Variables (Stand-Alone Authenticator To Lower Layer)

eapReq (boolean)

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

eapFail (boolean)

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

eapTimeout (boolean)

Set to TRUE in the TIMEOUT_FAILURE state if the authenticator has reached its maximum number of retransmissions without receiving a response.

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 define the structure of the type "EAP key". We expect that it will be defined in [Keying].

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 Stand-Alone Authenticator State Machine and Methods

IN: eapRespData, methodState

OUT: ignore, eapReqData

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

The following describes the interaction between the state machine and EAP methods.

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)

Next, the method creates a new EAP Request packet, with the given identifier value, and updates its method-specific state accordingly.

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

The method can also provide a hint for retransmission timeout with m.getTimeout.

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

When a new EAP Response is received, the method must first decide whether to process the packet or to discard it silently. If the packet looks like it was not sent by the legitimate peer (e.g., if it has an invalid Message Integrity Check (MIC), which 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 to 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. Stand-Alone 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 that 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.

ignore (boolean)

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

decision (enumeration)

Set in SELECT_ACTION state. Temporarily stores the policy decision to succeed, fail, or continue.

5.4. EAP Stand-Alone Authenticator Procedures

NOTE: For method procedures, the method uses its internal state in addition to the information provided by the EAP layer. The only arguments that are explicitly shown as inputs to the procedures are those provided to the method by EAP. Those inputs provided by the method's internal state remain implicit.

calculateTimeout()

Calculates the retransmission timeout, taking into account the retransmission count, round-trip time measurements, and method-specific timeout hint (see [RFC3748], Section 4.3). Returns an integer.

parseEapResp()

Determines the code, identifier value, and type of the current response. In the case of a parsing error (e.g., the length field is longer than the received packet), rxResp will be set to FALSE. The values of respId and respMethod may be undefined as a result. Returns a boolean, an integer, and an EAP type.

buildSuccess()

Creates an EAP Success Packet. Returns an EAP packet.

buildFailure()

Creates an EAP Failure Packet. Returns an EAP packet.

nextId()

Determines the next identifier value to use, based on the previous one. Returns an integer.

Policy.update()

Updates all variables related to internal policy state. The return value is undefined.

Policy.getNextMethod()

Determines the method that should be used at this point in the conversation based on predefined policy. Policy.getNextMethod() MUST comply with [RFC3748] (Section 2.1), which forbids the use of sequences of authentication methods within an EAP conversation. Thus, if an authentication method has already been executed within an EAP dialog, Policy.getNextMethod() MUST NOT propose another authentication method within the same EAP dialog. Returns an EAP type.

Policy.getDecision()

Determines if the policy will allow SUCCESS, FAIL, or is yet to determine (CONTINUE). Returns a decision enumeration.

m.check()

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

m.process()

Method procedure to parse and process a response for that method. The return value is undefined.

m.init()

Method procedure to initialize state just before use. The return value is undefined.

m.reset()

Method procedure to indicate that the method is ending in the middle of or before completion. The return value is undefined.

m.isDone()

Method procedure to check for method completion. Returns a boolean.

m.getTimeout()

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

m.getKey()

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

m.buildReq()

Method procedure to produce the next request. Returns an EAP packet.

5.5. EAP Stand-Alone 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

Between methods, the state machine re-evaluates whether 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 because no response has been received. Because no response was received, no new message (including failure) should be sent to the peer. Note that this is different from the FAILURE state, in which a message indicating failure is sent to the peer.

6. EAP Backend Authenticator

When operating in pass-through 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 that no retransmit is included in the IDLE state because with RADIUS, retransmit is handled by the NAS. Also, a PICK_UP_METHOD state and variable in INITIALIZE state are added to allow the Method to "pick up" 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. Note that the "lower layer" in this case is some AAA protocol (e.g., RADIUS).

(see the .pdf version for missing diagram or
refer to Appendix A.3 if reading the .txt version)

Figure 5: EAP Backend Authenticator State Machine

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. Usually indicates that an EAP response, stored in aaaEapRespData, is available for processing by the AAA server. If aaaEapRespData is set to NONE, it indicates that the AAA server should send the initial EAP request.

aaaEapRespData (EAP packet)

Set in lower layer when eapResp is set to TRUE. The EAP packet to be processed, or NONE.

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 that a new EAP request is ready to be sent.

aaaEapNoReq (boolean)

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

aaaSuccess (boolean)

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

aaaFail (boolean)

Set to TRUE in authenticator state machine, FALSE in lower layer. Indicates that 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 define the structure of the type "EAP key". We expect that it will be defined in [Keying].

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. (Note that this hint is for the EAP retransmissions done by the pass-through authenticator, not for retransmissions of AAA packets.)

6.2. Interface between Backend Authenticator State Machine and Methods

The backend method interface is almost the same as in stand-alone 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 pass-through. That is, the EAP Request packet was sent by the pass-through, 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 stand-alone version, as well as those that are defined differently.

NOTE: For method procedures, the method uses its internal state in addition to the information provided by the EAP layer. The only arguments that are explicitly shown as inputs to the procedures are those provided to the method by EAP. Those inputs provided by the method's internal state remain implicit.

Policy.doPickUp()

Notifies the policy that an already-chosen method is being picked up and will be completed. Returns a boolean.

m.initPickUp()

Method procedure to initialize state when continuing from an already-started method. The return value is undefined.

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 stand-alone version, as well as those that are defined differently.

PICK_UP_METHOD

Sets an initial state for a method that is being continued and that 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 stand-alone 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 it shows how the state machine works once it goes to pass-through mode.

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

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

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

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

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 pass-through 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. Note that there is not necessarily a direct interaction between the EAP layer and the AAA layer, as in the case of [1X-2004].

7.1.1. Variables (AAA Interface to Full Authenticator)

aaaEapReq (boolean)

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

aaaEapNoReq (boolean)

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

aaaSuccess (boolean)

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

aaaFail (boolean)

Set to TRUE in lower layer. Indicates that 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).

aaaEapKeyData (EAP key)

Set in lower layer when keying material becomes available from the AAA server. Note that this document does not define the structure of the type "EAP key". We expect that it will be defined in [Keying].

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. (Note that this hint is for the EAP retransmissions done by the pass-through authenticator, not for retransmissions of AAA packets.)

7.1.2. Variables (full authenticator to AAA interface)

aaaEapResp (boolean)

Set to TRUE in authenticator state machine, FALSE in the lower layer. Indicates that 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 has not 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 stand-alone 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 stand-alone version, as well as those that are defined differently.

7.3.1. Short-Term (Not Maintained between Packets)

decision (enumeration)

Set in SELECT_ACTION state. Temporarily stores the policy decision to succeed, fail, continue with a local method, or continue in pass-through mode.

7.4. EAP Full Authenticator Procedures

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

getId()

Determines the identifier value chosen by the AAA server for the current EAP request. The return value is an integer.

7.5. EAP Full Authenticator States

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

INITIALIZE_PASSTHROUGH

Initializes variables when the pass-through 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 that tells the AAA layer that 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 that 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 because no response has been received. Because no response was received, no new message (including failure) should be sent to the peer. Note that this is different from the FAILURE2 state, in which a message indicating failure is sent to the peer.

8. Implementation Considerations

8.1. Robustness

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 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 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.

8.2. Method/Method and Method/Lower-Layer Interfaces

Implementations may define additional interfaces to pass method-specific information between methods and lower layers. These interfaces are beyond the scope of this document.

8.3. Peer State Machine Interoperability with Deployed Implementations

Number of deployed EAP authenticator implementations, mainly in RADIUS authentication servers, have been observed to increment the Identifier field incorrectly when generating EAP Success and EAP Failure packets which is against the MUST requirement in RFC 3748 section 4.2. The peer state machine is based on RFC 3748, and as such it will discard such EAP Success and EAP Failure packets.

As a workaround for the potential interoperability issue with existing implementations, conditions for peer state machine transitions from RECEIVED state to SUCCESS and FAILURE states MAY be changed from "(reqId == lastId)" to "((reqId == lastId) || (reqId == (lastId + 1) & 255))". However, because this behavior does not conform to RFC 3748, such a workaround is not recommended, and if included, it should be implemented as an optional workaround that can be disabled.

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.

An accurate state machine can help reduce implementation errors. Although [RFC3748] remains the normative protocol description, this state machine should help in this regard.

As noted in [RFC3748], some security concerns arise because of the following EAP packets:

1. EAP-Request/Response Identity
2. EAP-Response/NAK
3. EAP-Success/Failure

Because these packets are not cryptographically protected by themselves, an attacker can modify or insert them without immediate detection by the peer or authenticator.

Following Figure 3 specification, an attacker may cause denial of service by:

- Sending an EAP-Failure to the peer before the peer has started an EAP authentication method. As long as the peer has not modified the methodState variable (initialized to NONE), the peer MUST accept an EAP-Failure.

- Forcing the peer to engage in endless EAP-Request/Response Identity exchanges before it has started an EAP authentication method. As long as the peer has not modified the selectedMethod variable (initialized to NONE), the peer MUST accept an EAP-Request/Identity and respond to it with an EAP-Response/Identity.

Following Figure 4 specification, an attacker may cause denial of service by:

- Sending a NAK to the authenticator after the authenticator first proposes an EAP authentication method to the peer. When the methodState variable has the value PROPOSED, the authenticator is obliged to process a NAK that is received in response to its first packet of an EAP authentication method.

There MAY be some cases when it is desired to prevent such attacks. This can be done by modifying initial values of some variables of the EAP state machines. However, such modifications are NOT RECOMMENDED.

There is a trade-off between mitigating these denial-of-service attacks and being able to deal with EAP peers and authenticators in general. For instance, if a NAK is ignored when it is sent to the authenticator after it has just proposed an EAP authentication method to the peer, then a legitimate peer that is not able or willing to process the proposed EAP authentication method would fail without an opportunity to negotiate another EAP method.

10. Acknowledgements

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 Meetinghouse Data Communications started independently with help from Dave Spence at Interlink Networks. John and Nick collaborated to create a common document, and then were joined by Pasi Eronen of Nokia, who has made major contributions in creating coherent state machines, and by Yoshihiro Ohba of Toshiba, who insisted on including pass-through documentation and provided significant support for understanding implementation issues.

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

11. References

11.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC3579] Aboba, B. and P. Calhoun, "RADIUS (Remote Authentication Dial In User Service) Support For Extensible Authentication Protocol (EAP)", RFC 3579, September 2003.

[RFC3748] Aboba, B., Blunk, L., Vollbrecht, J., Carlson, J., and H. Levkowetz, Ed., "Extensible Authentication Protocol (EAP)", RFC 3748, June 2004.

11.2. Informative References

[Keying] Aboba, B., Simon, D., Arkko, J., Eronen, P., Levkowetz, H., "Extensible Authentication Protocol (EAP) Key Management Framework", Work in Progress, July 2005.

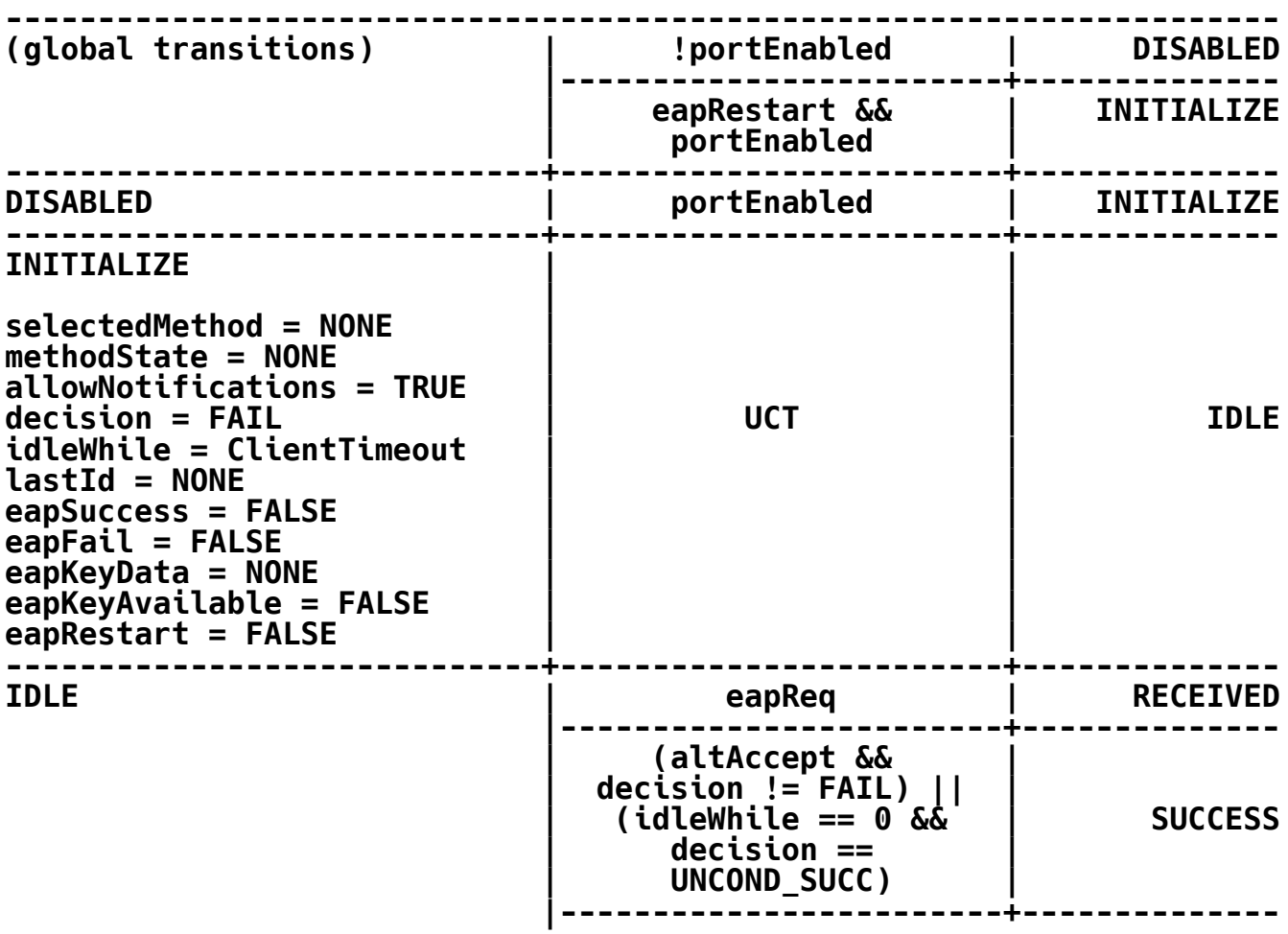
[1X-2004] Institute of Electrical and Electronics Engineers, "Standard for Local and Metropolitan Area Networks: Port-Based Network Access Control", IEEE 802.1X-2004, December 2004.

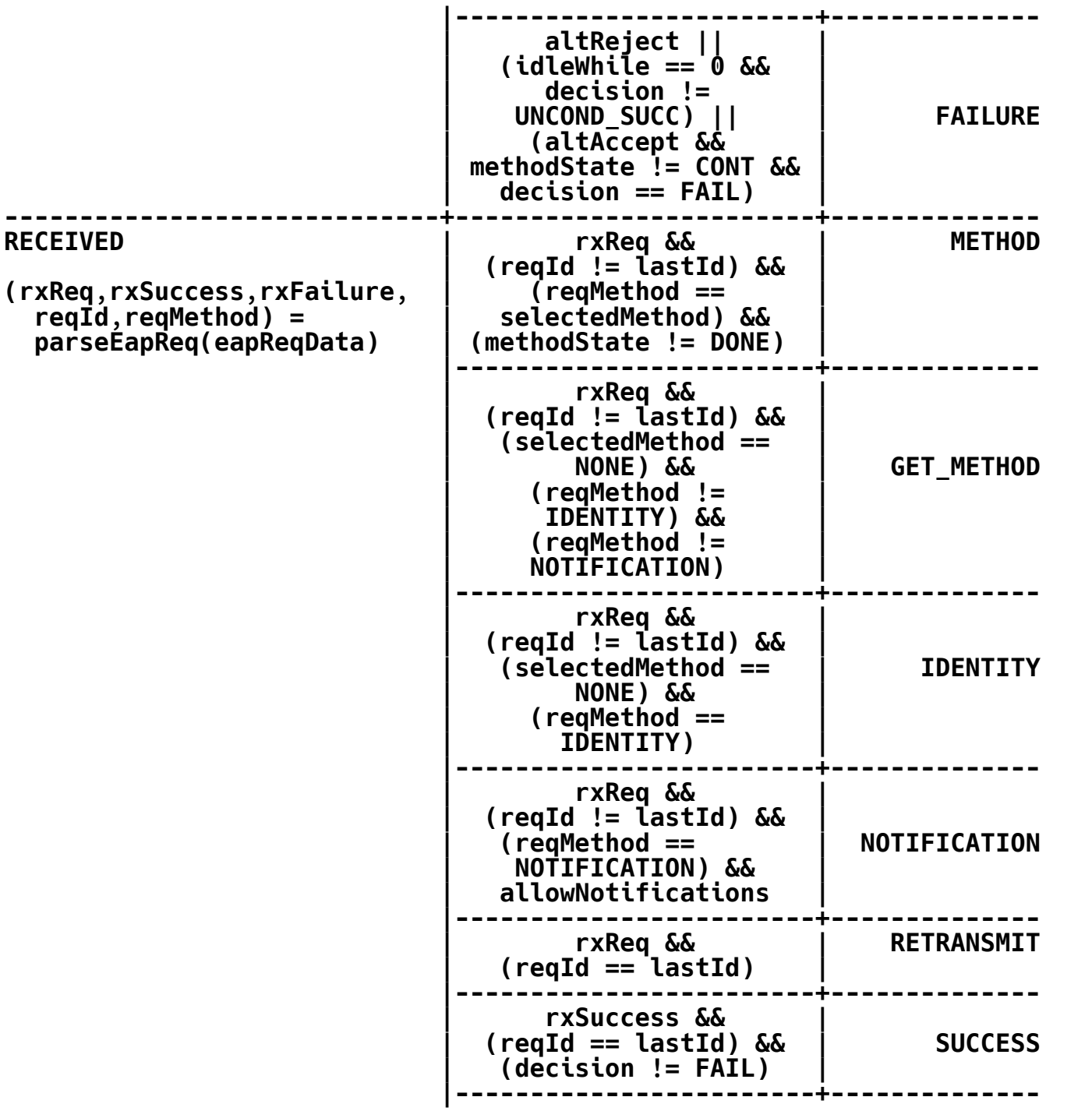
Appendix A. ASCII versions of state diagrams

This appendix contains the state diagrams in ASCII format. Please use the PDF version whenever possible; it is much easier to understand.

The notation is as follows: state name and pseudocode executed when entering it are shown on the left; outgoing transitions with their conditions are shown on the right.

A.1. EAP Peer State Machine (Figure 3)




altReject ||
(idleWhile == 0 &&
decision !=
UNCOND_SUCC) ||
(altAccept &&
methodState != CONT &&
decision == FAIL)
FAILURE
RECEIVED
(rxReq,rxSuccess,rxFailure,
reqId,reqMethod) =
parseEapReq(eapReqData)
rxReq &&
(reqId != lastId) &&
(reqMethod ==
selectedMethod) &&
(methodState != DONE)
METHOD
rxReq &&
(reqId != lastId) &&
(selectedMethod ==
NONE) &&
(reqMethod !=
IDENTITY) &&
(reqMethod !=
NOTIFICATION)
GET_METHOD
rxReq &&
(reqId != lastId) &&
(selectedMethod ==
NONE) &&
(reqMethod ==
IDENTITY)
IDENTITY
rxReq &&
(reqId != lastId) &&
(reqMethod ==
NOTIFICATION) &&
allowNotifications
NOTIFICATION
rxReq &&
(reqId == lastId)
RETRANSMIT
rxSuccess &&
(reqId == lastId) &&
(decision != FAIL)
SUCCESS

	(methodState!=CONT) && ((rxFailure && decision != UNCOND_SUCC) || (rxSuccess && decision == FAIL)) && (reqId == lastId)	FAILURE
	else	DISCARD

METHOD
 	
ignore = m.check(eapReqData)
if (!ignore) {
(methodState, decision,
allowNotifications) =
m.process(eapReqData)
/* methodState is CONT, MAY_CONT, or DONE */
/* decision is FAIL, COND_SUCC, or UNCOND_SUCC */
eapRespData =
m.buildResp(reqId)
if (m.isKeyAvailable())
eapKeyData = m.getKey()
}
	ignore	DISCARD

	(methodState==DONE) && (decision == FAIL)	FAILURE
	else	SEND_RESPONSE

GET_METHOD
 	
if (allowMethod(reqMethod)) {
selectedMethod = reqMethod
methodState = INIT
} else {
eapRespData =
buildNak(reqId)
}
	selectedMethod == reqMethod	METHOD

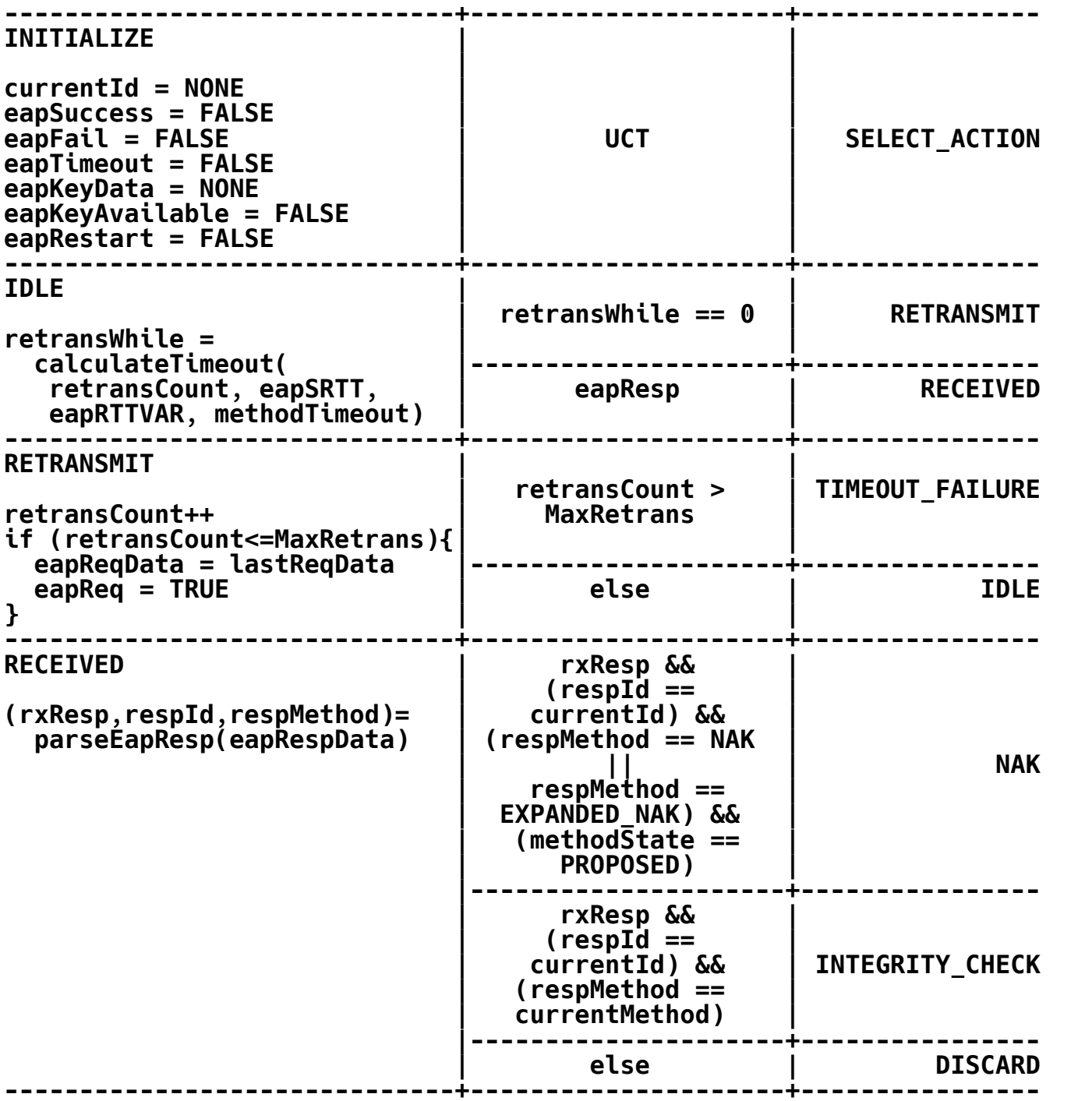
	else	SEND_RESPONSE
IDENTITY processIdentity(eapReqData) eapRespData = buildIdentity(reqId)	UCT	SEND_RESPONSE

NOTIFICATION processNotify(eapReqData) eapRespData = buildNotify(reqId)	UCT	SEND_RESPONSE
RETRANSMIT eapRespData = lastRespData	UCT	SEND_RESPONSE
DISCARD eapReq = FALSE eapNoResp = TRUE	UCT	IDLE
SEND_RESPONSE lastId = reqId lastRespData = eapRespData eapReq = FALSE eapResp = TRUE idleWhile = ClientTimeout	UCT	IDLE
SUCCESS if (eapKeyData != NONE) eapKeyAvailable = TRUE eapSuccess = TRUE		
FAILURE eapFail = TRUE		

Figure 8

A.2. EAP Stand-Alone Authenticator State Machine (Figure 4)

(global transitions)	!portEnabled	DISABLED
	eapRestart && portEnabled	INITIALIZE
DISABLED	portEnabled	INITIALIZE


INITIALIZE
currentId = NONE
eapSuccess = FALSE
eapFail = FALSE
eapTimeout = FALSE
eapKeyData = NONE
eapKeyAvailable = FALSE
eapRestart = FALSE
UCT
SELECT_ACTION
IDLE
retransWhile =
calculateTimeout(
retransCount, eapSRTT,
eapRTTVAR, methodTimeout)
retransWhile == 0
RETRANSMIT
eapResp
RECEIVED
RETRANSMIT
retransCount++
if (retransCount<=MaxRetrans){
eapReqData = lastReqData
eapReq = TRUE
}
retransCount > MaxRetrans
TIMEOUT_FAILURE
else
IDLE
RECEIVED
(rxResp,respId,respMethod)=
parseEapResp(eapRespData)
rxResp && (respId == currentId) && (respMethod == NAK || respMethod == EXPANDED_NAK) && (methodState == PROPOSED)
NAK
rxResp && (respId == currentId) && (respMethod == currentMethod)
INTEGRITY_CHECK
else
DISCARD

NAK m.reset() Policy.update(<...>)	UCT	SELECT_ACTION

SELECT_ACTION
 	
decision =
  Policy.getDecision()
/* SUCCESS, FAILURE, or
  CONTINUE */
	decision == FAILURE	FAILURE

	decision == SUCCESS	SUCCESS
	else	PROPOSE_METHOD
INTEGRITY_CHECK ignore = m.check(eapRespData)	ignore	DISCARD
	!ignore	METHOD_RESPONSE

METHOD_RESPONSE
 	
m.process(eapRespData)
if (m.isDone()) {
  Policy.update(<...>)
  eapKeyData = m.getKey()
  methodState = END
} else
  methodState = CONTINUE
	methodState == END	SELECT_ACTION

	else	METHOD_REQUEST

PROPOSE_METHOD
 	
currentMethod =
  Policy.getNextMethod()
m.init()
if (currentMethod==IDENTITY ||
  currentMethod==NOTIFICATION)
  methodState = CONTINUE
else
  methodState = PROPOSED
	UCT	METHOD_REQUEST

METHOD_REQUEST currentId = nextId(currentId) eapReqData = m.buildReq(currentId) methodTimeout = m.getTimeout()	UCT	SEND_REQUEST

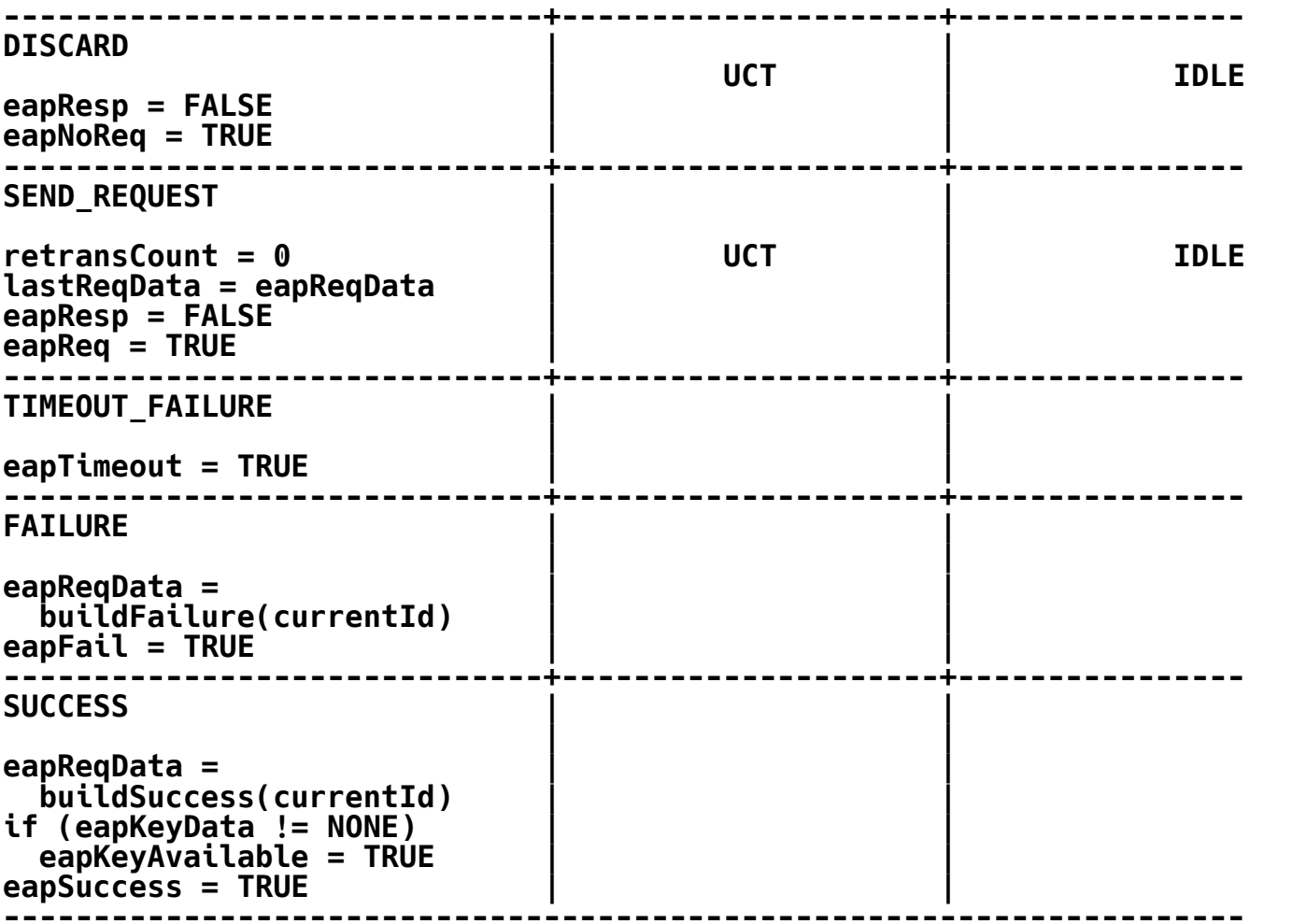
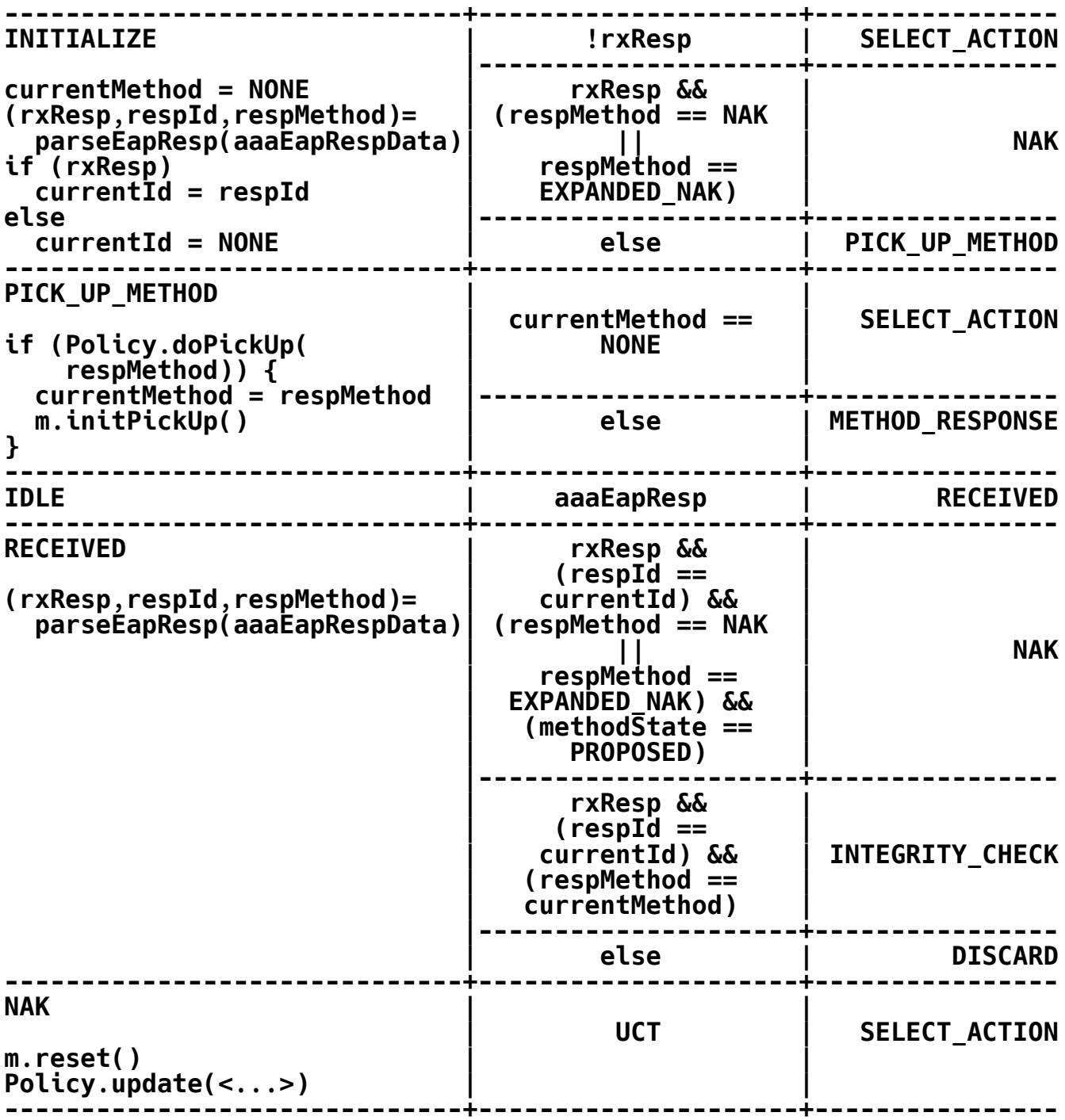


Figure 9

A.3. EAP Backend Authenticator State Machine (Figure 5)

(global transitions)	!backendEnabled	DISABLED
DISABLED	backendEnabled && aaaEapResp	INITIALIZE


INITIALIZE
currentMethod = NONE
(rxResp,respId,respMethod)=
parseEapResp(aaaEapRespData)
if (rxResp)
currentId = respId
else
currentId = NONE
!rxResp
SELECT_ACTION
rxResp &&
(respMethod == NAK
||
respMethod ==
EXPANDED_NAK)
NAK
else
PICK_UP_METHOD
PICK_UP_METHOD
if (Policy.doPickUp(
respMethod)) {
currentMethod = respMethod
m.initPickUp()
}
currentMethod ==
NONE
SELECT_ACTION
else
METHOD_RESPONSE
IDLE
aaaEapResp
RECEIVED
RECEIVED
(rxResp,respId,respMethod)=
parseEapResp(aaaEapRespData)
rxResp &&
(respId ==
currentId) &&
(respMethod == NAK
||
respMethod ==
EXPANDED_NAK) &&
(methodState ==
PROPOSED)
NAK
rxResp &&
(respId ==
currentId) &&
(respMethod ==
currentMethod)
INTEGRITY_CHECK
else
DISCARD
NAK
m.reset()
Policy.update(<...>)
UCT
SELECT_ACTION

SELECT_ACTION
 	
decision =
  Policy.getDecision()
/* SUCCESS, FAILURE, or
  CONTINUE */
	decision == FAILURE	FAILURE

	decision == SUCCESS	SUCCESS
	else	PROPOSE_METHOD
INTEGRITY_CHECK ignore = m.check(aaaEapRespData)	ignore	DISCARD
	!ignore	METHOD_RESPONSE

METHOD_RESPONSE
 	
m.process(aaaEapRespData)
if (m.isDone()) {
  Policy.update(<...>)
  aaaEapKeyData = m.getKey()
  methodState = END
} else
  methodState = CONTINUE
	methodState == END	SELECT_ACTION

	else	METHOD_REQUEST

PROPOSE_METHOD
 	
currentMethod =
  Policy.getNextMethod()
m.init()
if (currentMethod==IDENTITY ||
  currentMethod==NOTIFICATION)
  methodState = CONTINUE
else
  methodState = PROPOSED
	UCT	METHOD_REQUEST

METHOD_REQUEST
 	
currentId = nextId(currentId)
aaaEapReqData =
  m.buildReq(currentId)
aaaMethodTimeout =
  m.getTimeout()
	UCT	SEND_REQUEST

DISCARD aaaEapResp = FALSE aaaEapNoReq = TRUE	UCT	IDLE

SEND_REQUEST aaaEapResp = FALSE aaaEapReq = TRUE	UCT	IDLE
FAILURE aaaEapReqData = buildFailure(currentId) aaaEapFail = TRUE		

SUCCESS
 	
aaaEapReqData =
  buildSuccess(currentId)
if (aaaEapKeyData != NONE)
  aaaEapKeyAvailable = TRUE
aaaEapSuccess = TRUE		

Figure 10

A.4. EAP Full Authenticator State Machine (Figures 6 and 7)

This state machine contains all the states from EAP stand-alone authenticator state machine, except that SELECT_ACTION state is replaced with the following:

SELECT_ACTION
 	
decision =
  Policy.getDecision()
/* SUCCESS, FAILURE, CONTINUE,
  or PASSTHROUGH */
	decision == FAILURE	FAILURE

	decision == SUCCESS	SUCCESS
	decision == PASSTHROUGH	INITIALIZE_PASSTHROUGH
	else	PROPOSE_METHOD

Figure 11

And the following new states are added:

INITIALIZE_PASSTHROUGH aaaEapRespData = NONE	currentId != NONE	AAA_REQUEST
	currentId == NONE	AAA_IDLE

IDLE2
 	
retransWhile =
  calculateTimeout(
  retransCount, eapSRTT,
  eapRTTVAR, methodTimeout)
	retransWhile == 0	RETRANSMIT2

	eapResp	RECEIVED2

RETRANSMIT2
 	
retransCount++
if (retransCount<=MaxRetrans){
  eapReqData = lastReqData
  eapReq = TRUE
}
	retransCount > MaxRetrans	TIMEOUT_FAILURE2

	else	IDLE2
RECEIVED2 (rxResp,respId,respMethod)= parseEapResp(eapRespData)	rxResp && (respId == currentId)	AAA_REQUEST
	else	DISCARD2
AAA_REQUEST if (respMethod == IDENTITY) { aaaIdentity = eapRespData aaaEapRespData = eapRespData	UCT	AAA_IDLE
AAA_IDLE aaaFail = FALSE aaaSuccess = FALSE aaaEapReq = FALSE aaaEapNoReq = FALSE aaaEapResp = TRUE	aaaEapNoReq	DISCARD2
	aaaEapReq	AAA_RESPONSE
	aaaTimeout	TIMEOUT_FAILURE2
	aaaFail	FAILURE2
	aaaSuccess	SUCCESS2
AAA_RESPONSE eapReqData = aaaEapReqData currentId = getId(eapReqData) methodTimeout = aaaMethodTimeout	UCT	SEND_REQUEST2

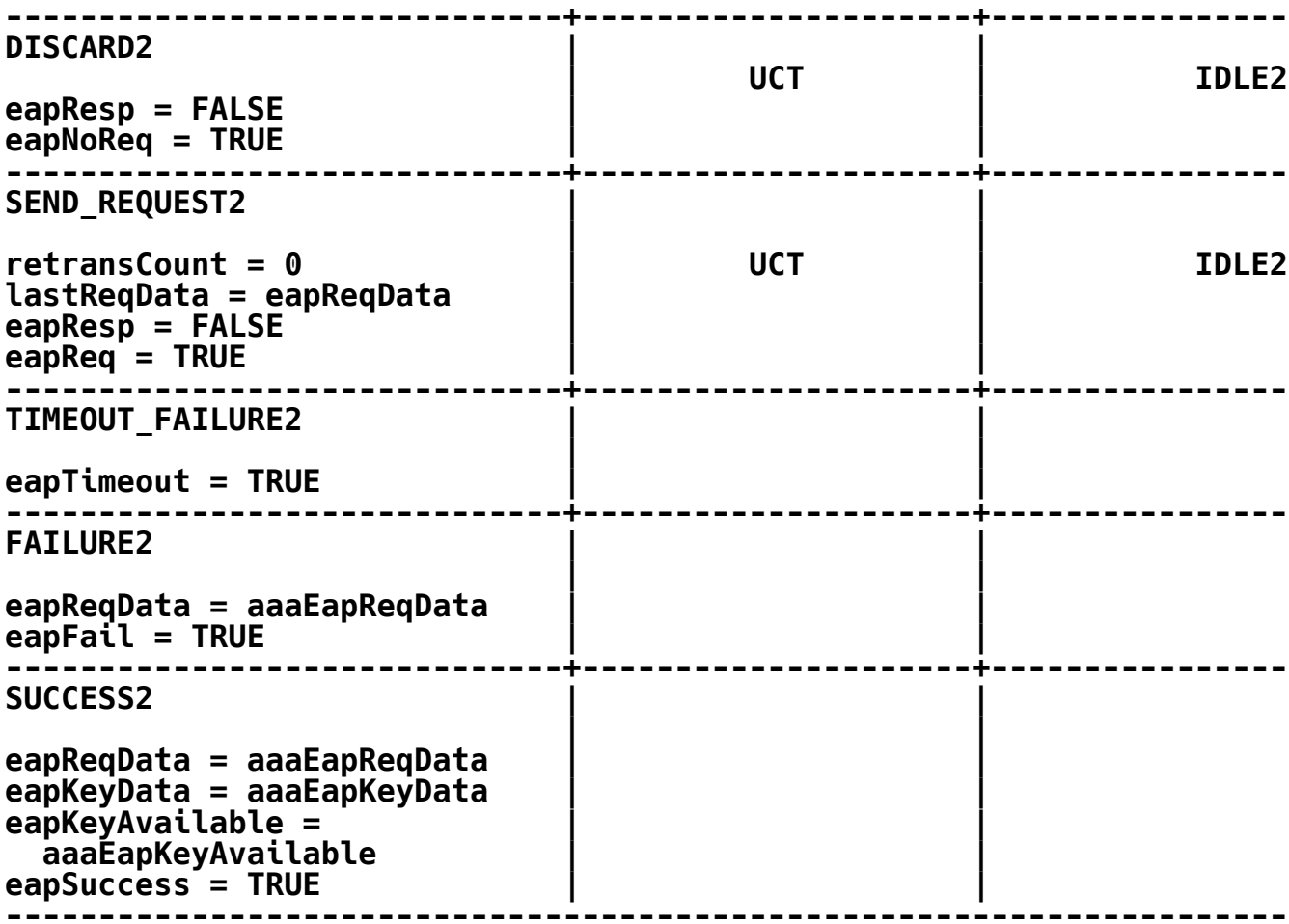

DISCARD2
eapResp = FALSE
eapNoReq = TRUE
UCT
IDLE2
SEND_REQUEST2
retransCount = 0
lastReqData = eapReqData
eapResp = FALSE
eapReq = TRUE
UCT
IDLE2
TIMEOUT_FAILURE2
eapTimeout = TRUE
FAILURE2
eapReqData = aaaEapReqData
eapFail = TRUE
SUCCESS2
eapReqData = aaaEapReqData
eapKeyData = aaaEapKeyData
eapKeyAvailable =
aaaEapKeyAvailable
eapSuccess = TRUE

Figure 12

Authors' Addresses

John Vollbrecht
Meetinghouse Data Communications
9682 Alice Hill Drive
Dexter, MI 48130
USA

EMail: jrv@mtghouse.com

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 Research, Inc.
1 Telcordia Drive
Piscataway, NJ 08854
USA

EMail: yohba@tari.toshiba.com

Full Copyright Statement

Intellectual Property

Acknowledgement

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