NSIS
Internet-Draft
Expires: May 2, 2005

C. Werner
X. Fu
Univ. Goettingen
H. Tschofenig
Siemens
C. Aoun
Nortel

November 2004

NSLP NAT/FW State Machine
draft-werner-nsis-natfw-nslp-statemachine-00.txt

Status of this Memo

Copyright Notice

Abstract

This document describes the state machines for the NSIS Signaling Layer Protocol for Network Address Translation/Firewall signaling (NAT/FW NSLP). A set of state machines for NAT/FW NSLP entities at

different locations of a signaling path are presented in order to illustrate how NAT/FW NSLP may be implemented.

Table of Contents

1. Introduction

This document describes the state machines for NAT/FW NSLP [1], trying to show how NAT/FW NSLP can be implemented to support its deployment. The state machines described in this document are illustrative of how the NAT/FW NSLP protocol defined in [1] may be implemented for the first NAT/FW NSLP node in the signaling path, intermediate NAT/FW NSLP nodes with Firewall and/or NAT functionality, and the last NAT/FW NSLP node in the signaling path. Where there are differences [1] are authoritative. The state machines are informative only. Implementations may achieve the same results using different methods.

The messages used in the NAT/FW NSLP protocol can be summarized as follows:

Requesting message	Responding message
CREATE	RESPONSE
REA	RESPONSE
QUERY	RESPONSE
RESPONSE	NONE
NOTIFY	NONE
TRIGGER	CREATE

We describe a set of state machines for different roles of entities running NAT/FW NSLP to illustrate how NAT/FW NSLP may be implemented.

2. Terminology

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 [2].

3. Notational conventions used in state diagrams

The following state transition tables are completed mostly based on the conventions specified in [3]. The complete text is described below.

State transition tables 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.

All permissible transitions from a given state to other states and associated actions performed when the transitions occur are represented by using triplets of (exit condition, exit action, exit state). All conditions are expressions that evaluate to TRUE or FALSE; if a condition evaluates to TRUE, then the condition is met. A state "ANY" is a wildcard state that matches the current state in each state machine. The exit conditions of a wildcard state are evaluated after all other exit conditions of specific to the current state are met.

On exit from a state, the procedures defined for the state and the exit condition are executed exactly once, in the order that they appear on the page. (Note that the procedures defined in [4] are executed on entry to a state, which is one major difference from this document.) Each procedure 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 the transition from the previous state to the current state, all exit conditions for the current state (including exit conditions defined for the wildcard state) are evaluated continuously until one of the conditions is met.

Any event variable is set to TRUE when the corresponding event occurs and set to FALSE immediately after completion of the action

associated with the current state and the event.

The interpretation of the special symbols and operators is reused from [4] and the state diagrams are based on the conventions specified in [5], 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.

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.

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.

4. 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 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.

> 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.

5. Common Rules

Throughout the document we use terms defined in the [1], such as NI, NF, NR, NI+, NR+, CREATE, QUERY, or RESPONSE.

5.1 Common Procedures

tx_CREATE(): Transmit a CREATE message
tx_CREATE(LIFETIME=0): Transmit CREATE message with lifetime object explicitly set to 0 for session deletion
tx_RESP(code,type): Transmit RESPONSE message with specified code (SUCCESS or ERROR) and result type (related to a specific request type message: CREATE, REA or QUERY). A code or result type may be omitted, typically when forwarding received RESPONSE messages.
tx_QUERY(): Transmit QUERY message.
tx_NOTIFY(): Transmit NOTIFY message.
rx_RESP(code, type): Evaluates to TRUE if a RESPONSE message has been received with the specified code (SUCCESS or ERROR) and result type (related to a specific request type message: CREATE, REA or QUERY). If the code or type is omitted, any received RESPONSE message which is only matching the given code or type will evaluate this procedure to TRUE.
rx_NOTIFY(): Evaluates to TRUE if a NOTIFY message has been received.
rx_QUERY(): Evaluates to TRUE if a QUERY message has been received
rx_CREATE(): Evaluates to TRUE if a CREATE message has been received.
CHECK_AA(): Checks Authorization and Authentication of the received message. Evaluates to TRUE if the check is successful, otherwise it evaluates to FALSE. This check is performed on all received messages hence it will only be shown within the state machine when the check has failed. This CHECK_AA also MAY include a local policy check for the received message.
CHECK_NoNR(): Checks if the message can reach its targeted destination, i.e. the NR if it exists at the targeted host.
CHECK_SCOPE(): Checks if the message has reached the network boundaries defined by the SCOPE object.
Process Event(): Processes a NOTIFY messages and adapts the behaviour of this node to the new condition.
Process Query(): Processes the received QUERY message and prepares the appropriate RESPONSE message.
Binding.create(): Creates a public/private network translation binding on a NAT device for the requesting entity.
Binding.clear(): Deletes a previously created a public/private network translation binding on a NAT device for the requesting

entity.
Session.create(): Installs all session related states, variables, bindings, policies.

Session.update(): Updates all session related states, variables, bindings, policies based on received CREATE or TRIGGER if applicable.
Session.clear(): Removes all session related states, variables, bindings, policies.
PckFilter.create(): Installs a packet filter for the new session.
PckFilter.update(): Updates the packet filter for changes in the session rules.
PckFilter.clear(): Removes a previously set packet filter.
Start.STATE_TIMER(identifier): This procedure starts a timer with a certain timespan, which is up to the specific implementation. The parameter 'identifier' identifies this timer uniquely. Any subsequent Start_STATE_TIMER(x), Stop_STATE_TIMER(x), TIMEOUT_STATE(x) refer to the same timer labeled x. This timer is required to time the lifetime of state, which means that when it times out, it indicates the current machine state should be left or its validation has expired. This procedure starts the timer 'identifier'. If a timer with the same 'identifier' has already been started and not yet stopped, the timer is now stopped and restarted. After the timer has timed out, the procedure TIMEOUT_STATE(identifier) evaluates to TRUE. The timer does not restart automatically, but must be started again with a Start_STATE_TIMER(identifier). Notice that there is no difference to the Start_REFRESH_TIMER(identifier) procedure which has exactly the same functionality. The different procedure names are only supplied to underline the purpose of this specific timer.
Stop.STATE_TIMER(identifier): This procedure stops the timer labeled 'identifier'. If it has already been stopped, this procedure has no effect. If the timer has already timed out, this procedure removes the timeout-state from the timer 'identifier', so subsequent calls to TIMEOUT_STATE(identifier) evaluate to FALSE. A timeout cannot occur until the timer 'identifier' has been (re-)started.
TIMEOUT.STATE(identifier): This procedure evaluates to TRUE if the timer 'identifier' has timed out and indicates a state lifetime expiration. Subsequent TIMEOUT_STATE(identifier) calls also

evaluate to TRUE until the timer 'identifier' has been (re-)started. This procedure cannot evaluate to TRUE if the timer has been stopped.

Start.REFRESH_TIMER(identifier): This procedure starts a timer with a certain timespan, which is up to the specific implementation. The parameter 'identifier' identifies this timer uniquely. Any subsequent Start_REFRESH_TIMER(x), Stop_REFRESH_TIMER(x), TIMEOUT_REFRESH(x) refer to the same timer labeled x. This timer times a refresh interval, which means that when it times out, it indicates a state refresh message is due to be sent. This procedure starts the timer 'identifier'. If a timer with the same 'identifier' has already been started and not yet stopped, the

timer is now stopped and restarted. After the timer has timed out, the procedure TIMEOUT_REFRESH(identifier) evaluates to TRUE. The timer does not restart automatically, but must be started again with a Start_REFRESH_TIMER(identifier). Notice that there is no difference to the Start_STATE_TIMER(identifier) procedure which has exactly the same functionality. The different procedure names are only supplied to underline the purpose of this specific timer.

Stop.REFRESH_TIMER(identifier): This procedure stops the timer labeled 'identifier'. If it has already been stopped, this procedure has no effect. If the timer has already timed out, this procedure removes the timeout-state from the timer 'identifier', so subsequent calls to TIMEOUT_REFRESH(identifier) evaluate to FALSE. A timeout cannot occur until the timer 'identifier' has been (re-)started.

TIMEOUT.REFRESH(identifier): This procedure evaluates to TRUE if the timer 'identifier' has timed out and indicates a refresh interval expiration. Subsequent TIMEOUT_REFRESH(identifier) calls also evaluate to TRUE until the timer 'identifier' has been (re-)started. This procedure cannot evaluate to TRUE if the timer has been stopped.

tg_QUERY: External trigger to send a QUERY message (typically triggered by the application).

tg_CREATE: External trigger to send a CREATE message (typically triggered by the application).

tg_NOTIFY: External trigger to notify the entity of a new event to be processed (typically triggered by the application)

tg_TRIGGER: External trigger to send a TRIGGER message to a NF (typically triggered by the application)

tg_TEARDOWN: External trigger to delete a previously created session (typically triggered by the application)

tg_REA: External trigger to send a REA message towards an opportunistic address (typically triggered by the application)

5.2 Common Variables

IS_EDGE: Boolean flag which evaluates to TRUE if the node is on the network edge, otherwise it evaluates to FALSE.

IS_PUBLICSIDE: Boolean flag which evaluates to TRUE if the (CREATE- or REA-) message has been received on the public side of the network.

CREATE(LIFETIME?): Gets the value of the LIFETIME object in the CREATE message.

CREATE(TRIGGER?): Evaluates to TRUE if the received CREATE message indicates a CREATE trigger.

CREATE(POLICY?): Gets the policy for the CREATE message.

CREATE(SOURCE?): Retrieves the sender of the CREATE message.

CREATE(NoNR?): Evaluates to TRUE if the CREATE message has an active NoNR-flag.

CREATE(Scope?): Evaluates to TRUE if the CREATE message has an active Scope-flag.

Retry_Counter(CREATE): Denotes the current number of retries of CREATE message which has been re-transmitted due to previous RESPONSE_ERROR message. If the number of Retry_Counter(CREATE) equals the value of MAXRETRY(CREATE), the current session creation attempt is aborted and the application is being notified.

Retry_Counter(QUERY): Denotes the current number of retries of QUERY message which has been re-transmitted due to previous RESPONSE_ERROR message. If the number of Retry_Counter(QUERY) equals the value of MAXRETRY(QUERY), the current QUERY attempt is aborted and the application is being notified.

Retry_Counter(REA): Denotes the current number of retries of REA message which has been re-transmitted due to previous RESPONSE_ERROR message. If the number of Retry_Counter(REA) equals the value of MAXRETRY(REA), the current REA initiation attempt is aborted and the application is being notified.

5.3 Constants

Max_Retry(CREATE): Contains the maximum number of retransmission attempts of a CREATE message after it is aborted and the application is being notified.
Max_Retry(QUERY): Contains the maximum number of retransmission attempts of a QUERY message after it is aborted and the application is being notified.
Max_Retry(REA): Contains the maximum number of retransmission attempts of a REA message after it is aborted and the application is being notified.

6. State machine for the NAT/FW NI

This section presents the state machines for the NSIS initator which is capable of NSLP NAT/FW signaling

State: Initialize

Condition	Action	State
UCT	retry_Counter(Create)=0; retry_Counter(Query)=0;	IDLE

```
-------------------
State: IDLE
-------------------

Condition                 Action                       State
------------------------+--------------------------+------------
tg_CREATE                |Start.STATE_TIMER(Resp); |PENDING
                         |retry_Counter(Create)=0; |
                         |tx_CREATE;               |
------------------------+--------------------------+------------
```



```
-------------------
State: PENDING
-------------------

Condition                 Action                       State
------------------------+--------------------------+------------
rx_RESP(SUCCESS,Create) |Stop.STATE_TIMER(Resp);  |ESTABLISHED
                         |Session.create();        |
```

```
                          |Start.REFRESH_TIMER(Cre);|
                          |retry_Counter(Create)=0; |
                          |                         |
TIMEOUT.STATE(Resp)       |Stop.STATE_TIMER(Resp);  |PENDING
                          |retry_Counter(Create)++; |
                          |if (retry_Counter(Create)|
                          |<=Max_Retry(Create))     |
                          |{Start.STATE_TIMER(Resp);|
                          |tx_CREATE;}              |
                          |                         |
(Retry_Counter(Create)    |Send info to appl.;      |IDLE
> Max_Retry(Create)) ||   |Stop.STATE_TIMER(Resp);  |
tg_TEARDOWN ||            |                         |
rx_RESP(ERROR,Create)     |                         |
--------------------------+-------------------------+------------


-------------------
State: ESTABLISHED
-------------------

Condition                  Action                     State
--------------------------+-------------------------+------------
rx_RESP(SUCCESS,Query)    |Stop.STATE_TIMER(Query); |ESTABLISHED
&& CHECK_AA               |Send info to appl.;      |
                          |                         |
tg_QUERY                  |tx_QUERY;                |ESTABLISHED
                          |Start.STATE_TIMER(Query);|
                          |retry_Counter(Query)=0;  |
                          |                         |
rx_RESP(SUCCESS,Create)   |Start.REFRESH_TIMER(Cre);|ESTABLISHED
                          |Stop.STATE_TIMER(Resp);  |
                          |retry_counter(Create)=0; |
                          |                         |
TIMEOUT.REFRESH(Cre)      |Start.STATE_TIMER(Resp); |ESTABLISHED
                          |tx_CREATE;               |
                          |                         |
TIMEOUT.STATE(Resp)       |Stop.STATE_TIMER(Resp);  |ESTABLISHED
                          |retry_Counter(Create)++; |
                          |if (retry_Counter(Create)|
```



```
                          | <= Max_Retry(Create)) { |
```

```
                        |Start.STATE_TIMER(Resp); |
                        |tx_CREATE;}              |
                        |                         |
rx_NOTIFY && CHECK_AA   |Process Event();         |ESTABLISHED
                        |                         |
rx_RESP(ERROR,Query) || |Stop.STATE_TIMER(Query); |ESTABLISHED
TIMEOUT.STATE(Query)    |retry_Counter(Query)++;  |
                        |if (retry_Counter(Query) |
                        | <= Max_Retry(Query)) {  |
                        |Start.STATE_TIMER(Query);|
                        |tx_QUERY;} else {        |
                        |send info to appl.}      |
                        |                         |
(retry_Counter(Create)  |Send info to appl.;      |IDLE
> Max_Rety(Create)) ||  |Session.clear();         |
rx_RESP(ERROR,Create)   |Stop.REFRESH_TIMER(Cre); |
                        |                         |
tg_TEARDOWN             |tx_CREATE(LIFETIME=0);   |IDLE
                        |Session.clear();         |
                        |Stop.REFRESH_TIMER(Cre); |
                        |Stop.STATE_TIMER(Resp);  |
------------------------+-------------------------+------------
```

7. State machines for the NAT/FW NF

This section describes the state machines for intermediate nodes within the signaling path capable of processing NAT/FW NSLP messages. These nodes typically implement firewall and/or network address translation (NAT) functionality. To keep it simple, the state machines are separated in two independent state machines for nodes with firewall and nodes with NAT functionality.

7.1 State machine for NAT/FW Firewall NF

State: Initialize

Condition	Action	State
UCT	-	IDLE

State: IDLE

Condition	Action	State
rx_REA && !(CHECK_AA)	tx_RESP(ERROR,Rea);	IDLE
rx_RESP(Rea)	tx_RESP(Rea);	IDLE
rx_REA && IS_EDGE	tx_RESP(ERROR,Rea); (*)	IDLE
rx_REA && !(IS_EDGE)	tx_REA;	IDLE
rx_CREATE && CHECK_AA	Start.STATE_TIMER(Resp); tx_CREATE;	PENDING
rx_CREATE && !(CHECK_AA)	tx_RESP(ERROR,Create);	IDLE

* REA Error message "No NAT here"

```
------------------
State: PENDING
------------------

Condition                 Action                      State
------------------------+--------------------------+------------
rx_RESP(SUCCESS,Create) |Stop_STATE_TIMER(Resp);  |ESTABLISHED
                        |Session.create();        |
                        |PckFilter.create();      |
                        |Start.STATE_TIMER(Cre);  |
                        |                         |
rx_RESP(ERROR,Create)   |Stop.STATE_TIMER(Resp);  |IDLE
|| TIMEOUT.STATE(Resp)  |                         |
------------------------+--------------------------+------------
```

State: ESTABLISHED

Condition	Action	State
rx_CREATE && !(CHECK_AA)	tx_RESP(ERROR,Create);	ESTABLISHED
rx_TRIGGER && CHECK_AA && !IS_EDGE	tx_TRIGGER;	ESTABLISHED
rx_RESP(SUCCESS,Create)	Start.STATE_TIMER(Cre); tx_RESP(SUCCESS,Create);	ESTABLISHED
rx_QUERY && CHECK_AA	Process Query(); tx_QUERY;	ESTABLISHED
rx_CREATE && CHECK_AA && CREATE(LIFETIME?)>0	tx_CREATE;	ESTABLISHED
rx_RESP(,Query) && CHECK_AA	tx_RESP(,Query);	ESTABLISHED
tg_NOTIFY	tx_NOTIFY;	ESTABLISHED
rx_NOTIFY && CHECK_AA	Process Event(); tx_NOTIFY;	ESTABLISHED
TIMEOUT.STATE(Cre) || tg_TEARDOWN	Session.clear(); PckFilter.clear();	IDLE
rx_CREATE && CHECK_AA	tx_CREATE(LIFETIME=0);	IDLE

```
&& CREATE(LIFETIME?)==0 |Session.clear();          |
                        |PckFilter.clear();        |
                        |Stop.STATE_TIMER(Cre);    |
------------------------+-------------------------+------------
```

7.2 State machine for NAT/FW NAT NF

```
-------------------
State: Initialize
-------------------

Condition                Action                     State
-------------------------+-------------------------+------------
        UCT              |Retry_Counter(Create)=0; |IDLE
```



```
------------------------+-------------------------+------------


-------------------
State: IDLE
-------------------

Condition                Action                     State
-------------------------+-------------------------+------------
rx_CREATE &&             |tx_RESP(ERROR,Rea);  (*1)|IDLE
IS_PUBLICSIDE            |                         |
                         |                         |
rx_REA && CHECK_AA &&    |tx_RESP(ERROR,Rea);  (*2)|IDLE
&& IS_PUBLICSIDE         |                         |
                         |                         |
rx_REA && CHECK_AA &&    |Binding.create();        |NonEDGE REA
!IS_EDGE                 |tx_REA;                  |
                         |Start_STATE_TIMER(Rea);  |
                         |                         |
rx_REA && CHECK_AA &&    |Binding.create();        |REA
IS_EDGE &&               |Start.STATE_TIMER(Rea);  |
!IS_PUBLICSIDE           |tx_RESP(SUCCESS,Rea);    |
                         |retry_Counter(Create)=0; |
                         |Start.STATE_TIMER(Resp); |
                         |tx_CREATE;               |
```

```
                          |                           |
rx_CREATE && CHECK_AA     |Binding.create();          |PENDING
&& !IS_PUBLICSIDE         |Start.STATE_TIMER(Resp);   |
                          |tx_CREATE;                 |
--------------------------+---------------------------+------------
*1 Error message is "No reservation made"
*2 Error message is "REA received on public side"
```



```
-------------------
State: NonEDGE REA
-------------------

Condition                  Action                      State
-------------------------+---------------------------+------------
rx_RESP(,Query) &&        |tx_RESP(,Query);           |NonEDGE REA
CHECK_AA                  |                           |
                          |                           |
rx_QUERY && CHECK_AA      |Process Query();           |NonEDGE REA
                          |tx_QUERY;                  |
                          |                           |
tx_TRIGGER && CHECK_AA    |PckFilter.update();        |NonEDGE REA
                          |Start.STATE_TIMER(Rea);    |
                          |tx_TRIGGER;                |
                          |                           |
rx_CREATE && CHECK_AA     |Stop.STATE_TIMER(Rea);     |PENDING
                          |Start.STATE_TIMER(Resp);   |
```

	|tx_CREATE;	|
	|	|
TIMEOUT.STATE(Rea) ||	|Binding.clear();	|IDLE
rx_RESPONSE(ERROR,Rea)	|PckFilter.clear();	|

State: REA

Condition	Action	State
TIMEOUT.STATE(Rea) ||	|Binding.clear();	|IDLE
rx_RESP(ERROR,Create)	|	|
	|	|
rx_RESP(SUCCESS,Create)	|Stop.STATE_TIMER(Resp);	|REA
	|	|

```
rx_CREATE &&              |Start.STATE_TIMER(Resp); |NI-PENDING
CREATE(SOURCE?)==NI &&    |tx_CREATE;               |
CHECK_AA                  |                         |
                          |                         |
TIMEOUT.STATE(Resp)       |Stop.STATE_TIMER(Resp);  |REA
                          |retry_Counter(Create)++; |
                          |if (retry_Counter(Create)|
                          | <= Max_Retry(Create)) { |
                          |tx_CREATE;               |
                          |Start.STATE_TIMER(Resp);}|
                          |                         |
rx_TRIGGER && CHECK_AA    |Start.STATE_TIMER(Rea);  |REA
                          |Start.STATE_TIMER(Resp); |
                          |retry_Counter(Create)=0; |
                          |PckFilter.update();      |
                          |tx_CREATE;               |
                          |                         |
tg_NOTIFY                 |tx_NOTIFY;               |REA
                          |                         |
rx_QUERY && CHECK_AA      |tx_RESPONSE(,Query);     |REA
--------------------------+-------------------------+------------
```

```
-------------------
State: PENDING
-------------------
```

```
Condition                  Action                     State
-------------------------+-------------------------+------------
rx_RESP(SUCCESS,Create)  |Stop.STATE_TIMER(Resp);  |ESTABLISHED
                         |Start.STATE_TIMER(Cre);  |
                         |Session.create();        |
                         |PckFilter.create();      |
                         |tx_RESP(SUCCESS,Create); |
                         |                         |
rx_RESP(ERROR,Create)    |Binding.remove();        |IDLE
|| TIMEOUT.STATE(Resp)   |                         |
```



```
--------------------------+-------------------------+------------
```

```
-------------------
```

State: NI-PENDING

Condition	Action	State
rx_RESP(SUCCESS,Create)	Stop.STATE_TIMER(Resp); Session.clear(); Session.create(); PckFilter.create(); Start.STATE_TIMER(Cre); tx_RESP(SUCCESS,Create);	ESTABLISHED
rx_RESP(ERROR,Create) || TIMEOUT.STATE(Resp)	-	REA

State: ESTABLISHED

Condition	Action	State
rx_CREATE && !(CHECK_AA)	tx_RESP(ERROR,Create);	ESTABLISHED
rx_QUERY && CHECK_AA	Process Query(); tx_QUERY;	ESTABLISHED
TIMEOUT.STATE(Cre) || tg_TEARDOWN	Session.clear(); send info to appl.;	IDLE
rx_CREATE && CHECK_AA && CREATE(LIFETIME?)==0	tx_CREATE(LIFETIME=0); Session.clear(); PckFilter.clear();	IDLE
rx_TRIGGER && !IS_EDGE && CHECK_AA	PckFilter.update(); tx_TRIGGER;	ESTABLISHED
rx_RESP(,Query) && CHECK_AA	tx_RESP(,Query);	ESTABLISHED
rx_NOTIFY && CHECK_AA	Process Event(); tx_NOTIFY;	ESTABLISHED
tg_NOTIFY	tx_NOTIFY;	ESTABLISHED
rx_CREATE && CHECK_AA && CREATE(LIFETIME?)>0	PckFilter.update(); tx_CREATE;	ESTABLISHED
rx_RESP(SUCCESS,Create)	Start.STATE_TIMER(Cre); tx_RESP(SUCCESS,Create);	ESTABLISHED
rx_RESP(ERROR,Create)	tx_RESP(ERROR,Create);	ESTABLISHED

8. State machine for the NAT/FW NR

This section presents the state machines for the NSIS responder which is capable of NSLP NAT/FW signaling

State: Initialize

Condition	Action	State
UCT (*)	Retry_Counter(REA)=0;	IDLE

* Triggered by application when forking process

State: IDLE

Condition	Action	State
rx_CREATE && CHECK_AA	tx_RESP(SUCCESS,Create); Start.STATE_TIMER(Cre); Session.start(); PckFilter.create(); Send info to appl.	ESTABLISHED
tg_REA	tx_REA; retry_Counter(REA)=0; Start.STATE_Timer(Resp);	REA PENDING
rx_CREATE && !(CHECK_AA)	tx_RESP(ERROR,Create);	IDLE

State: REA PENDING

Condition	Action	State
rx_RESP(SUCCESS,Rea)	Stop.STATE_TIMER(Resp); Start.STATE_TIMER(Cre);	TRIG PENDING

TIMEOUT.STATE(Resp)	retry_Counter(REA)++;
	if (retry_Counter(REA) <= Max_Retry(REA)) {
	Start.STATE_TIMER(Resp);
	tx_REA;}
		REA PENDING

(retry_Counter(REA) > Max_Retry(REA)) || rx_RESP(ERROR,Rea)	Send info to appl.;	IDLE

State: TRIG PENDING

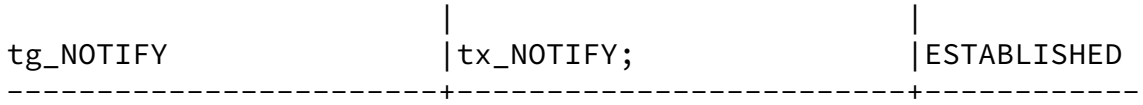
Condition	Action	State
TIMEOUT.STATE(Cre)	Send info to appl.;	IDLE
rx_CREATE && CHECK_AA	tx_RESP(SUCCESS,Create); Session.create(); PckFilter.create(); Send info to appl.;	ESTABLISHED

```
                          |Start.STATE_TIMER(Cre);  |
                          |Start.REFRESH_TIMER(Trg);|
------------------------+-------------------------+------------
```



```
------------------
State: ESTABLISHED
------------------

Condition                Action                      State
------------------------+-------------------------+------------
(rx_CREATE &&            |Session.session();       |IDLE
CREATE(LIFETIME?)==0 &&  |PckFilter.clear();       |
CHECK_AA) ||             |Send info to appl.;      |
TIMEOUT.STATE(Cre) ||    |                         |
tg_TEARDOWN              |                         |
                         |                         |
TIMEOUT.REFRESH(Trg)     |tx_TRIGGER;              |ESTABLISHED
                         |Start.REFRESH_TIMER(Trg);|
                         |                         |
rx_QUERY && CHECK_AA     |Process Query();         |ESTABLISHED
                         |tx_RESP(,Query);         |
                         |                         |
rx_CREATE &&             |if (CREATE(SOURCE?)!=NF){|ESTABLISHED
CREATE(LIFETIME?)>0 &&   |Stop.REFRESH_TIMER(Trg);}|
CHECK_AA                 |tx_RESP(SUCCESS,Create); |
                         |Start.STATE_TIMER(Cre);  |
                         |                         |
rx_CREATE && !(CHECK_AA)|tx_RESP(ERROR,Create);    |ESTABLISHED
                         |                         |
rx_NOTIFY && CHECK_AA    |Process Event();         |ESTABLISHED
```



9. Security Considerations

This document does not raise new security considerations. Any security concerns with the NAT/FW NSLP are likely reflected in security related NSIS work already (such as [1] or [6]).

For the time being, the state machines described in this document do not consider the security aspect of NAT/FW NSLP protocol itself. A future version of this document will add security relevant states and state transitions.

10. Open Issues

CREATE[NoNR] and CREATE[Scope] message triggers are currently not implemented in the state machines and all other open issues in [1] will be added in future versions of this document.

11. Acknowledgments

The authors would like to thank Tseno Tsenov for his valuable comments and discussions.

12. References

12.1 Normative References

[1] Stiemerling, M., "A NAT/Firewall NSIS Signaling Layer Protocol (NSLP)", draft-ietf-nsis-nslp-natfw-04 (work in progress), October 2004.

[2] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", March 1997.

12.2 Informative References

[3] Ohba, Y., "State Machines for Protocol for Carrying Authentication for Network Access (PANA)", draft-ohba-pana-statemachine-00 (work in progress), July 2004.

[4] Vollbrecht, J., Eronen, P., Petroni, N. and Y. Ohba, "State Machines for Extensible Authentication Protocol (EAP) Peer and Authenticator", draft-ietf-eap-statemachine-05 (work in progress), September 2004.

[5] Institute of Electrical and Electronics Engineers, "DRAFT Standard for Local and Metropolitan Area Networks: Port-Based Network Access Control (Revision)", IEEE 802-1X-REV/D9, January 2004.

[6] Tschofenig, H. and D. Kroeselberg, "Security Threats for NSIS", draft-ietf-nsis-threats-06 (work in progress), October 2004.

Authors' Addresses

Constantin Werner
University of Goettingen
Telematics Group
Lotzestr. 16-18
Goettingen 37083
Germany

EMail: werner@cs.uni-goettingen.de

Xiaoming Fu
University of Goettingen
Telematics Group
Lotzestr. 16-18
Goettingen 37083
Germany

EMail: fu@cs.uni-goettingen.de

Hannes Tschofenig
Siemens
Otto-Hahn-Ring 6
Munich, Bayern 81739
Germany

EMail: Hannes.Tschofenig@siemens.com

Cedric Aoun
Nortel Networks/ENST Paris

EMail: cedric.aoun@nortelnetworks.com

Intellectual Property Statement

Disclaimer of Validity

Copyright Statement

Acknowledgment

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