SCTP NAT Automatic Test Utilities

David Hayes, Jason But
Centre for Advanced Internet Architectures, Technical Report 081128B
Swinburne University of Technology
Melbourne, Australia
david.hayes@ieee.org, jbut@swin.edu.au

***Abstract*—The development of the SCTP NAT automatic test utilities is part of the SONATA[1] project to develop and release a BSD licensed implementation of a Network Address Translation (NAT) module that supports the Stream Control Transmission Protocol (SCTP). The test utilities work with master/slave architecture communicating over a control channel. Tests are specified in configuration files on the master host. This report gives instructions on how tests can be specified, with examples of common SCTP NAT functionality tests.**

I. Introduction

As part of the development of the FreeBSD kernel SCTP NAT implementation, `alias_sctp`, some versatile automated functionality testing utilities have been developed. In this report we outline the architecture of these test utilities and how to develop configuration files to perform various testing scenarios using these utilities.

The NAT is a black box that translates IP address of packets traversing the private and public sides of the device. Typically this would involve multiple hosts on each side of the NAT. It is however feasible to test NAT functionality using a single device on either side of the NAT and configuring tests to emulate conditions that would normally only be seen with multiple devices.

We employ a master/slave architecture – tests are completely managed by the master executing on a host on the local side of the NAT. Tests are designed and written as configuration files to be processed by the master.

We also note that the SCTP NAT test suite is not specific to our `alias_sctp` implementation and has been specifically developed to be used with any SCTP NAT implementation.

This report is structured as follows, in Section II we discuss the architecture used in the testing environment while Section III looks at the details of executing the Python scripts on the test platforms. In Section IV we provide details on the formats of the test configuration files with some examples provided in Section V. In the Appendices we provide some further examples along with a listing of library requirements for the test applications.

II. Automatic test architecture

A NAT typically handles connections from and to multiple clients. However to test the NAT's functionality it is not necessary to re-construct this physical layout. It is possible to perform many functionality tests using a single client on either side of the NAT. The range of tests that can be performed is greatly expanded if the host on the global side of the NAT is multi-homed. Further testing is possible if both local and remote hosts are multi-homed.

In our test architecture we initiate testing from the local side of the NAT, since access from the global side depends on port forwarding rules installed in the NAT. A slave utility is first launched on the global side which waits for a connection from the master (local side of the NAT). The master establishes a control channel (see Figure 1) through which it can control the execution of commands on the slave. A series of test channels are then established as required to complete the programmed tests. The tests to be performed are outlined in a set of configuration files which are read and parsed by the master utility.

A. Address configuration

Both the local and global hosts must have a configuration file (.cfg), that contains their IP address and network interface information (see section IV for details on these files). After establishment of the control channel, the local host sets the global host's response timeout (*resp_TO*) variable via the control channel. It is important for both hosts to agree on the response timeout to avoid synchronisation problems.

B. Test execution

The local host reads its test procedures from one or more configuration files (.cfg). Each section defines a

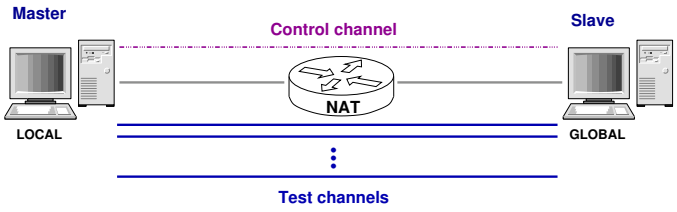

Fig. 1. Automatic test setup

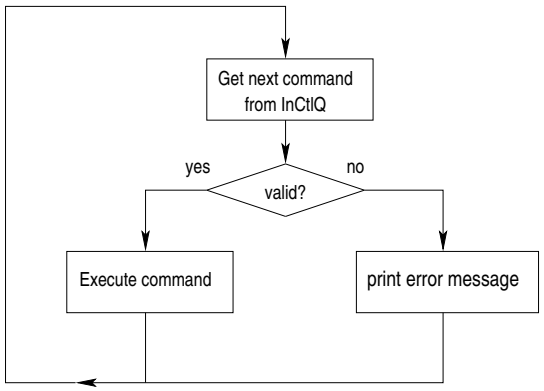

Fig. 3. Slave (global) basic operation

test where instructions within the test are executed in sequential order. Test sections are executed in alphabetical order. The basic operation of the master for a particular test section is shown in figure 2. Instructions can be either commands for the master (local host) or the slave (global host), or to set variables on the master. Operations on the master side are checked for validity and then executed. If there are any errors, a message is printed, and the master attempts to continue processing.

Instructions for the slave are not checked for validity, but sent over the control channel. The slave simply receives commands from the master, checks their validity, and executes them (see figure 3).

The success or failure of instructions on the master's side is tracked, with a summary of the pass rate for each test (instructions within a test section), and the overall pass rate when all tests have been completed. The results of instructions that pass are output in green, while those that fail are printed in red. Any problems in the execution of the instructions are printed in blue.

C. Channel Queues

Packets are received via the pcap interface (python *Pcapy* module[2]). A thread is started to look after the incoming packets on each interface the host has defined in its configuration file.

All packets received on any of the interfaces are placed in either the `InCtlQ` or the `InPktQ` queues

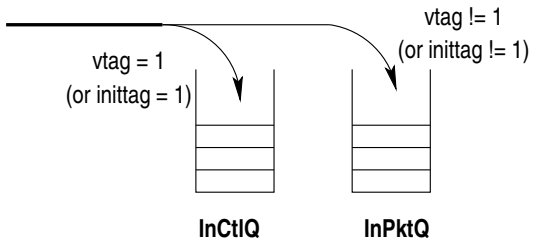

Fig. 4. Incoming packets

(see figure 4). The control channel is discriminated using $vtags = ports = 1$ values. Packets arriving on the control channel are placed in the `InCtlQ`. All other packets are placed in the `InPktQ` queues. The queueing utilises the python *Queue*, which handles the necessary mutex thread safe locking. When either the local host, or the global host is waiting for a response it issues a **blocking get** request to the respective queue with a *resp_TO* timeout.

It is also necessary to install a firewall rule on each end-host to discard SCTP packets. This prevents the local SCTP stack from receiving and responding to the packets generated during testing.

III. Python scripts

The automatic SCTP NAT utilities consist of:

- `RawSctp.py` – SCTP encoding and decoding classes (based on the Impacket library[3])
- `sctpNATtestCommon.py` – Common classes for both the master and slave scripts.
- `sctpNATtestL` – Executable **master** script, run on the local side of the NAT.
- `sctpNATtestG` – Executable **slave** script, run on the global side of the NAT.

Command line options for `sctpNATtestL` and `sctpNATtestG` are shown in Code segments 1 and 2 respectively. The only compulsory option is for `sctpNATtestL`, which requires a destination IP address on the slave host.

IV. Configuration file

The auto test utilities rely on configuration files ("`.cfg`") files to define their IP addresses and the tests to be conducted. The names of the files are not significant. The test utilities will load all "`.cfg`" files in the current directory (or the one given by the path option).

The basic format of the configuration files is: `[section heading]` followed by instructions. Lines that begin with "#" are comments.

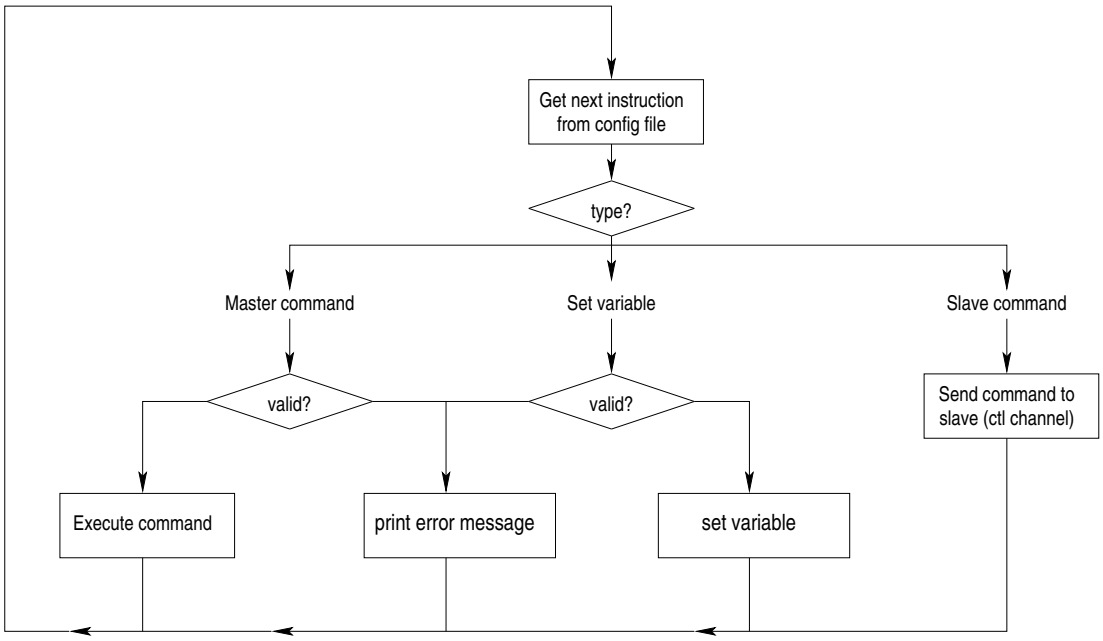

Fig. 2. Master (local) basic operation

Code segment 1 Local host (master) command line options

```
Options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  -v, --verbose         turns on verbose mode, showing all command states and
                        packets received
  -n, --nocolour        turns colour mode on
  -d ip_address, --dest=ip_address
                        destination ip address of slave (global host)
  -t N, --timeout=N     timout after N seconds when waiting for responses
                        [default: 2]
  -p PATH, --config_path=PATH
                        path to config files (.cfg)
```

Code segment 2 Global host (slave) command line options

```
Options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  -v, --verbose         turns on verbose mode, showing all command states and
                        packets received
  -w N, --wait=N        Wait N seconds for commands from master [default: 300]
  -p PATH, --config_path=PATH
                        path to config files (.cfg)
```

A. Address configuration

Both the master and slave hosts must have a configuration file that describes their network interfaces and corresponding IP addresses. The section is headed by the title "`[MY_ADDRESSES]`". Each item under this section heading consists of label and a tuple of $< interface, address >$. See code segment 3 for an example.

Code segment 3 Example address Configuration

```
[MY_ADDRESSES]
myip1=('tun0','10.0.1.2')
myip2=('tun1','10.0.2.2')
```

There should not be multiple "`[MY_ADDRESSES]`" configuration sections within the configuration files, since they will overwrite each other.

B. Test configuration

Only the master uses the test configuration files. Test configuration files consist of a section heading "`[Test Name]`" for each test, followed by a set of *Instructions*. Tests are run in alphabetical order, while the instructions are stepped through in the order given in the configuration file. Instructions consist of an identifier, followed by a ":". There are four identifiers for instructions:

- `Mcmd:` *command to be executed on the master host*
- `Scmd:` *command to be sent to the slave host for execution*
- `TestRtn:` *test the returned result of the last instruction*
- `SetVar:` *set a variable on the master host*

Commands and their descriptions are shown in Table I for the master, and Table II for the slave.

Three variables may be set using the `SetVar` identifier:

- `CurrentSrc` – the default source address to use when sending an SCTP test packet. This variable defaults to *srcaddrs[0]*.
- `CurrentDst` – the default destination address to use when sending an SCTP test packet. This variable defaults to *dstaddr*
- `CurrentPayload` – the default payload data to be put into a test **DATA** chunk. This variable defaults to *'Test Data'*.

V. Examples

This section will look at two test configuration examples in detail.

Code segment 4 shows the configuration for a simple multi-homed setup. The test proceeds as follows:

1) First an association instance labeled *MT* is created. This single command creates the instance on both the master and the slave side hosts.
2) The slave is then instructed to wait for an **INIT** chunk on association *MT*.
3) The Master then sends the **INIT** and implicitly waits for an **INIT-ACK**
4) The master then initiates a test of sending data on all source↔destination address pairs, instructing the slave to echo the data back, checking that what was sent is received.
5) The slave is instructed to wait for a **SHUTDOWN-ACK** chunk, and respond with a **SHUTDOWN-COMPLETE** chunk.
6) The master sends a **SHUTDOWN-ACK** chunk and waits for the **SHUTDOWN-COMPLETE**.

Code segment 4 Example to test all paths in a multi-homed setup

```
[Multi-home Test]
Mcmd: SetAssoc MT (10,10,10,10)
Scmd: WaitInit MT
Mcmd: Init MT
Mcmd: TestAllPaths MT
Scmd: WaitClose MT
Mcmd: Close MT
```

The second example shown in Code segment 5 tests the response of the NAT when it receives a packet from an unknown association. It also demonstrates the **SetVar**, **TestRtn**, and the different **TestPath** methods. The `SetVar` examples are not necessary in this case, since they are setting the variables to their default values. They are here as an example.

The test proceeds as follows:

1) An association instance, ID *TE*, is created.
2) Set variable *CurrentSrc* to *srcaddrs[0]*. The configuration files are permitted to use variables that are internal to the python scripts. In this case *srcaddrs* is a list of the master's addresses.
3) Set variable *CurrentDst* to *dstaddr*. In this case *dstaddr* is the address given on the master's command line.
4) Set variable *CurrentPayload* to `'Test Data'`.
5) Print to stdout a comment to say what the test is about
6) If the NAT is working properly the next command should fail. `FailisPass` tells the master to treat

Master command	Description
`SetAssoc` *assocID* `(sport,dport,rx_vtag,tx_vtag)`	Establish a new association instance on the master and the slave identified by *assocID*. The tuple of source port, destination port, vtag for receiving, vtag for transmitting, is from the master's perspective. An association must be defined in this way, before it is referenced by any of the other commands. *Note: this does not establish an association between the master and the slave, that is done by the* `Init` *or* `ootbAddAddr` *commands*
`Init` *assocID*	Initialise the association identified by *assocID* (this must have been defined prior to this with the `SetAssoc` command). The master will send an **INIT** to the slave, and wait for an **INIT-ACK**
`WaitInit` *assocID*	Wait for initialisation of the association identified by *assocID*. That is, it will wait for an **INIT** from the slave, and respond with an **INIT-ACK**
`TxData` *assocID*	Transmit data on the association identified by *assocID* using the *CurrentSrc* as the source IP address, *CurrentDst* as the current destination IP address, and *CurrentPayload* as the payload for the **DATA** chunk. *Note: It is not necessary to have an established SCTP association to issue this command. It can be used to check the NATs response to Out Of The Blue (OOTB) packets.*
`RxData` *assocID*	Receive data on the association identified by *assocID*.
`RxAny` *assocID*	Receive any SCTP message on the association identified by *assocID*.
`AddAddr` *assocID* `newaddress`	Uses an **ASCONF-AddIP** to add the given address to the association. It will then wait for an **ASCONF-ACK**. The **ASCONF-AddIP** is sent using the base source and destination addresses.
`ootbAddAddr` *assocID* `newaddress`	Similar to `AddAddr`, except the source address will be `newaddress`.
`WaitAddAddr` *assocID*	Wait for an **ASCONF-AddIP** on from association *assocID* and respond with an **ASCONF-ACK**.
`DelAddr` *assocID* `ip_address`	Uses an **ASCONF-DelIP** to delete the given address to the association. It will then wait for an **ASCONF-ACK**.
`WaitDelAddr` *assocID*	Wait for an **ASCONF-DelIP** on from association *assocID* and respond with an **ASCONF-ACK**.
`Close` *assocID*	Send a **SHUTDOWN-ACK** on association *assocID*, and wait for a **SHUTDOWN-COMPLETE**.
`WaitClose` *assocID*	Wait for a **SHUTDOWN-ACK** on association *assocID*, and respond with a **SHUTDOWN-COMPLETE**.
`Abort` *assocID*	Send an **ABORT** on on association *assocID*.
`TestPath` *assocID* `srcaddr dstaddr payload`	Test the source↔destination path using `payload` as the data in the **DATA** chunk. The master sends `payload` to the slave, and the slave echos back whatever it received. The master then compares what it sent with what it received. A match is a pass. Normally the `TestPath` command performs the equivalent of: 1) Scmd: `RxData` 2) Mcmd: `TxData` 3) Mcmd: `RxData` 4) Scmd: `EchoData`
`TestCurrentPath` *assocID*	Similar to `TestPath` except *CurrentSrc*, *CurrentDst*, and *CurrentPayload* are used.
`TestAllPaths` *assocID*	This command does a `TestPath` for all source↔destination address pairs defined for this association.
`RxSlaveCtlData`	Receive data from the slave on the control channel. This is useful if the master wants to find out the result of a failed slave command.
`FailisPass`	The result of the next command will be treated as a pass if it fails. This is useful when testing the response of the NAT to OOTB packets, which depending on the configuration should not pass through the NAT. A pass for the NAT, is then the failure of the command.
`Wait` N	Wait N seconds, before proceeding.
`Print` '*text*'	Print the given text to stdout. Useful for adding comments to aid in interpretation of the results.
`PrintLastRxChunk` *assocID*	Decode and print the last received chunk on *assocID* to stdout.
`FlushTestQ`	Remove any queued packets in the test packet queue.

TABLE I
COMMANDS THE MASTER INTERPRETS

Slave command	Description
`SetRespTO` *N*	Set the packet wait timeout to *N*, where *N* can be a decimal number. The master issues this command to the slave during initialisation, so that both master's response wait time and the slave's are the same. *It should not be present in the configuration file.*
`SetAssoc` *assocID* `(sport,dport,rx_vtag,tx_vtag)`	This command is sent by the master as part of the action it performs when it is issued the command. *It should not be present in the configuration file.*
`Init` *assocID*	As for master.
`WaitInit` *assocID*	As for master.
`TxData` *assocID*	As for master.
`RxData` *assocID*	As for master.
`EchoData` *assocID*	This requests the slave to send a **DATA** chunk containing the contents of the last **DATA** chunk it recievied on the association identified by *assocID*.
`AddAddr` *assocID* `newaddress`	As for master.
`ootbAddAddr` *assocID* `newaddress`	As for master.
`WaitAddAddr` *assocID*	As for master.
`DelAddr` *assocID* `ip_address`	As for master.
`WaitDelAddr` *assocID*	As for master.
`Close` *assocID*	As for master.
`WaitClose` *assocID*	As for master.
`Abort` *assocID*	As for master.
`TxLastChunkInfo` *assocID*	Transmit the decoded last chunk received on *assocID* to the master on the control channel (in a **DATA** chunk)
`FlushTestQ`	As for Master.
`WaitCtlClose`	Wait for the control channel to close. This is automatically issued by the master after all the tests are complete. *It should not be present in the configuration file.*

TABLE II
COMMANDS THE SLAVE INTERPRETS

the fail as a pass.

7) Test association *TE* by sending data, having the slave echo what it received, and checking the received value.
8) Set the instruction return value to be the value of the last SCTP chunk received on *TE*.
9) Test the return value, to check that it is *'Error'*. In this case, if the NAT is responding to local side OOTB packets with an **ERROR-M** chunk, the last chunk we received should be an error.
10) Wait for the slave to time out waiting for data it wasn't going to receive. The internal variable *resp_TO* is used. This parameter defaults to 2 seconds, but is configurable on the command line through the `--timeout` option.
11) Tell the slave to wait for an **ASCONF-AddIP** on *TE* and respond with an **ASCONF-ACK**.
12) Send an OOTB **ASCONF-AddIP** chunk from *srcaddrs[0]*, adding *srcaddrs[0]*, and wait for the **ASCONF-ACK**.
13) Test association *TE* by sending *TestCurrentPayload* using *CurrentSrc* and *CurrentDst*, having the slave echo what it received, and checking the received value. This is the same as the previous test, except that now we expect it to pass.
14) This does the same as the above, except that it specifies the source address to be *srcaddrs[0]*, the destination address to be *dstaddr*, and the payload of the **DATA** chunk as `'Payload'`.
15) This is another way of doing the same thing, this time specifying the source and destination addresses `'192.168.0.65'` and `'10.0.1.2'`.
16) Instruct the slave to wait for the association to close down. That is wait for a **SHUTDOWN-ACK**

and respond with a **SHUTDOWN-COMPLETE**.

17) Send a **SHUTDOWN-ACK** and wait for a **SHUTDOWN-COMPLETE**.

A. Writing configuration files

Some points to remember when writing configuration files:

- All control channel and test packets are queued.
  - This means that it is often fine for one side to send something before the other side is asked to receive it. However it is good practice not to do this.
  - The slave gets one instruction from the control queue, processes it, and then gets the next one. When an instruction waiting for a response does not get it, it will eventually time out, and only then will the next instruction be executed. As such care must be taken to ensure that the slave will be ready to respond. Code segment 5 gives an example of this.
- In a multi-homed setup, packets will travel on different paths and may not arrive in the order they are sent. We recommend adding "`Mcmd: Wait 1`" between instructions where this may occur.
- The `FlushTestQ` command can be useful in restoring the InPktQ to the empty state. This is automatically done at the beginning of each test.
  - Be aware that if the `FlushTestQ` command is sent to the slave, it will be put on the InCtlQ, and be processed once all other previous instructions have been processed. For this reason, sometimes it may be necessary to add a `Wait` after issuing this command.

VI. Conclusion

The SCTP NAT automatic testing utilities provide a framework for testing the functionality of an SCTP NAT. The tests are conducted via a master/slave arrangement via a control channel. New tests can be specified in configuration files on the master (local side) host, without any new test programs being written.

Acknowledgements

The development of the SCTP NAT automatic test utilities is part of the SONATA [1] project and was made possible in part by a grant from the Cisco University Research Program Fund at Community Foundation Silicon Valley.

The project benefits from the people and facilities of CAIA.

References

[1] CAIA, "SONATA SCTP over NAT adaptation," viewed 12 June 2008. [Online]. Available: http://caia.swin.edu.au/urp/sonata

[2] C. S. Technologies, "Pcapy," viewed on 28 November 2008. [Online]. Available: http://oss.coresecurity.com/projects/pcapy.html

[3] ——, "Impacket," viewed on 28 November 2008. [Online]. Available: http://oss.coresecurity.com/projects/impacket.html

Appendix A
Additional examples

Two additional examples are included to aid those who wish to specify their own tests. The test shown in code segment 6 tests how the NAT handle's lookup table conflicts. The test shown in code segment 7 tests the NAT functionality when addresses are added to and removed from an association (which is important when global IP addresses are tracked).

Appendix B
Useful internal variables

Table III gives a list of useful internal variables that can be addressed in configuration files. Refer to the different configuration files to see how they can be used.

Code segment 5 Example to test response of a NAT to an unknown association

```
[Test Error AddIP]
Mcmd: SetAssoc TE (11,11,11,11)
SetVar: CurrentSrc = srcaddrs[0]
SetVar: CurrentDst = dstaddr
SetVar: CurrentPayload = 'Test Data'
Mcmd: Print 'Try sending when there is no associaion up in the NAT'
Mcmd: FailisPass
Mcmd: TestCurrentPath TE
Mcmd: LastRxChunkInfo TE
TestRtn: 'Error'
#wait for slave to timout on RxData
Mcmd: Wait resp_TO*2
Scmd: WaitAddAddr TE
Mcmd: ootbAddAddr TE srcaddrs[0]
Mcmd: Print 'Run a few test paths'
#shorthand testpath - uses CurrentSrc, CurrentDst, and CurrentPayload
Mcmd: TestCurrentPath TE
#long shorthand testpath
Mcmd: TestPath TE srcaddrs[0] dstaddr 'Payload'
#or
Mcmd: TestPath TE '192.168.0.65' '10.0.1.2' 'Payload'
Scmd: WaitClose TE
Mcmd: Close TE
```

Internal Variable	Description
srcaddrs[]	list of the master's ip addresses, as given in the configuration file
dstaddr	slave destination address, as given on the command line
resp_TO	Time to wait for a response, before timing out

TABLE III
USEFUL INTERNAL VARIABLES THAT CAN BE ADDRESSED IN CONFIGURATION FILES

Code segment 6 Example to test the response of the NAT to look up table collisions

```
[Test Init-Abort]
Mcmd: SetAssoc TA1 (33,33,33,33)
Scmd: WaitInit TA1
Mcmd: Init TA1
Mcmd:  Print 'Try to cause an abort, by using same vtags and ports'
Mcmd: SetAssoc TA2 (33,33,33,33)
Scmd: WaitInit TA2
Mcmd: FailisPass
Mcmd: Init TA2
Mcmd: LastRxChunkInfo TA2
TestRtn: 'Abort'
# wait for slave timout
Mcmd: Wait resp_TO*2
Mcmd: Print 'Try to cause an abort,'
Mcmd: Print '  with only the vtag the global host will put on packets different'
Mcmd: SetAssoc TA3 (33,33,44,33)
Scmd: WaitInit TA3
Mcmd: FailisPass
Mcmd: Init TA3
Mcmd: LastRxChunkInfo TA3
TestRtn: 'Abort'
# wait for slave timout
Mcmd: Wait resp_TO*2
Mcmd: Print 'Try to cause an abort,'
Mcmd: Print '   with only the vtag the local host will put on packets different'
Mcmd: SetAssoc TA4 (33,33,33,44)
Scmd: WaitInit TA4
Mcmd: FailisPass
Mcmd: Init TA4
Mcmd: LastRxChunkInfo TA4
TestRtn: 'Abort'
# wait for slave timout
Mcmd: Wait resp_TO*2
Mcmd: Print 'Only the local src port different'
Mcmd: Print '   is enough since match in both directions is src/dst ports'
Mcmd: SetAssoc TA5 (44,33,33,33)
Scmd: WaitInit TA5
Mcmd: Init TA5
Mcmd: Print 'Only the dst port different'
Mcmd: Print '   is enough since match in both directions is src/dst ports'
Mcmd: SetAssoc TA6 (33,44,33,33)
Scmd: WaitInit TA6
Mcmd: Init TA6
Mcmd:  Print 'Try a valid connection with different vtags'
Mcmd: SetAssoc TA7 (33,33,44,44)
Scmd: WaitInit TA7
Mcmd: Init TA7
Scmd: WaitClose TA1
Mcmd: Close TA1
Scmd: WaitClose TA5
Mcmd: Close TA5
Scmd: WaitClose TA6
Mcmd: Close TA6
Scmd: WaitClose TA7
Mcmd: Close TA7
```

Code segment 7 Example to test adding and removing IP addresses for an association that is established through the NAT

```
[Test DelIP]
Mcmd: SetAssoc TD (22,22,22,22)
Scmd: WaitInit TD
Mcmd: Init TD
Mcmd: TestAllPaths TD
#delete the last of the list of source addresses
Mcmd: Print 'Deleting 10.0.2.2 and test path'
Scmd: DelAddr TD '10.0.2.2'
Mcmd: WaitDelAddr TD
#need to wait here otherwise the AsconfAck may beat
#   the test data packet (they travel on different paths)
Mcmd: Wait 1
Mcmd: FailisPass
Mcmd: TestPath TD srcaddrs[0] '10.0.2.2' 'Payload 1'
#wait for slave RxData to time out
Mcmd: Wait resp_TO*2
Mcmd: Print 'Adding 10.0.2.2 and test path'
Scmd: AddAddr TD '10.0.2.2'
Mcmd: WaitAddAddr TD
#need to wait here otherwise the AsconfAck may beat
#   the test data packet (they travel on different paths)
Mcmd: Wait 1
Mcmd: TestPath TD srcaddrs[0] '10.0.2.2' 'Payload 2'
Scmd: WaitClose TD
Mcmd: Close TD
```