
SIPp Documentation

Release 3.6

SIPp community

Jun 19, 2019

Contents:

1	Foreword	1
2	Installation	3
2.1	Getting SIPp	3
2.2	SIPp releases	3
2.3	Unstable release	3
2.4	Available platforms	4
2.5	Installing SIPp	4
3	Main features	7
3.1	Running SIPp in background	7
3.2	Screens	7
3.3	Exit codes	10
3.4	Contributing to SIPp	10
4	Integrated scenarios	11
4.1	UAC	11
4.2	UAC with media	11
4.3	UAS	12
4.4	regex	12
4.5	branch	13
4.6	UAC Out-of-call Messages	13
4.7	3PCC	14
5	Create your own XML scenarios	17
5.1	Create your own XML scenarios	17
5.1.1	List of attributes common to all commands	19
5.1.2	List of commands with their attributes	20
5.1.3	Structure of client (UAC like) XML scenarios	27
5.1.4	Structure of server (UAS like) XML scenarios	28
5.2	Keyword list	29
5.2.1	[service]	29
5.2.2	[remote_ip]	29
5.2.3	[remote_port]	29
5.2.4	[transport]	29
5.2.5	[local_ip]	29
5.2.6	[local_ip_type]	30

5.2.7	[local_port]	30
5.2.8	[len]	30
5.2.9	[call_number]	30
5.2.10	[cseq]	30
5.2.11	[call_id]	30
5.2.12	[media_ip]	30
5.2.13	[media_ip_type]	30
5.2.14	[media_port]	30
5.2.15	[auto_media_port]	31
5.2.16	[last_*]	31
5.2.17	[field0-n file=<filename> line=<number>]	31
5.2.18	[file name=<filename>]	31
5.2.19	[timestamp]	31
5.2.20	[last_message]	31
5.2.21	[\$n]	31
5.2.22	[authentication]	31
5.2.23	[pid]	32
5.2.24	[routes]	32
5.2.25	[next_url]	32
5.2.26	[branch]	32
5.2.27	[msg_index]	32
5.2.28	[cseq]	32
5.2.29	[clock_tick]	32
5.2.30	[sipp_version]	32
5.2.31	[tdmmap]	32
5.2.32	[fill]	32
5.2.33	[users]	33
5.2.34	[userid]	33
5.3	Actions	33
5.3.1	Regular expressions	33
5.3.2	Log a message	35
5.3.3	Execute a command	35
5.3.4	Internal commands	35
5.3.5	External commands	36
5.3.6	Media/RTP commands	36
5.3.7	Variable Manipulation	37
5.3.8	String Variables	38
5.3.9	Variable Testing	38
5.3.10	lookup	38
5.3.11	Updating In-Memory Injection files	39
5.3.12	Jumping to an Index	39
5.3.13	gettimeofday	40
5.3.14	setdest	40
5.3.15	verifyauth	40
5.4	Variables	41
5.5	Injecting values from an external CSV during calls	42
5.5.1	PRINTF Injection files	43
5.6	Printf Injection File Parameters	43
5.6.1	Indexing Injection files	44
5.7	Conditional branching	44
5.7.1	Conditional branching in scenarios	44
5.7.2	Randomness in conditional branching	44
5.8	SIP authentication	45
5.9	Initialization Stanza	47

6	3PCC Extended	49
7	Controlling SIPp	51
7.1	List of Interactive Commands	51
7.2	Traffic control	52
7.3	Remote control	53
8	Transport modes	55
8.1	UDP mono socket	55
8.2	UDP multi socket	55
8.3	UDP with one socket per IP address	55
8.4	TCP mono socket	56
8.5	TCP multi socket	56
8.6	TCP reconnections	56
8.7	TLS mono socket	56
8.8	TLS multi socket	57
8.9	SCTP mono socket	57
8.10	SCTP multi socket	57
8.11	IPv6 support	57
8.12	Multi-socket limit	57
9	Handling media with SIPp	59
9.1	RTP echo	59
9.2	RTP streaming	59
9.3	PCAP Play	59
10	Statistics	61
10.1	Response times	61
10.2	Available counters	61
10.3	Detailed Message Counts	63
11	Error handling	65
11.1	Unexpected messages	65
11.2	Retransmissions (UDP only)	65
11.3	Log files	66
12	Performance testing with SIPp	67
12.1	Advice to run performance tests with SIPp	67
12.2	SIPp's internal scheduling	67
13	Useful tools aside SIPp	69
13.1	JEdit	69
13.2	Wireshark/tshark	69
13.3	SIP callflow	69
14	Indices and tables	71
	Index	73

CHAPTER 1

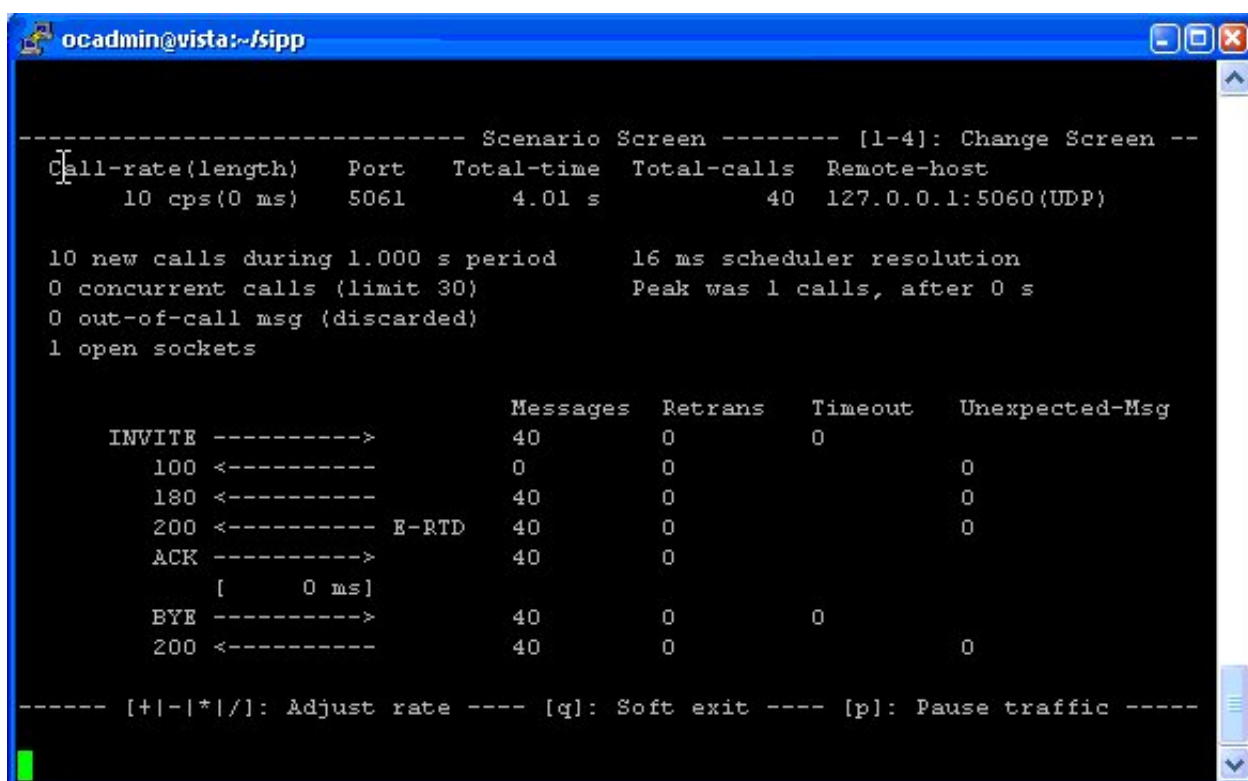
Foreword

Warning This version of the documentation is for SIPp 3.6 and describes some features not present in earlier versions. See the sidebar to access documentation for previous versions. SIPp is a performance testing tool for the SIP protocol. It includes a few basic SipStone user agent scenarios (UAC and UAS) and establishes and releases multiple calls with the INVITE and BYE methods. It can also read XML scenario files describing any performance testing configuration. It features the dynamic display of statistics about running tests (call rate, round trip delay, and message statistics), periodic CSV statistics dumps, TCP and UDP over multiple sockets or multiplexed with retransmission management, regular expressions and variables in scenario files, and dynamically adjustable call rates.

SIPp can be used to test many real SIP equipments like SIP proxies, B2BUAs, SIP media servers, SIP/x gateways, and SIP PBXes. It is also very useful to emulate thousands of user agents calling your SIP system.

Want to see it?

Here is a screenshot



```

ocadmin@vista:~/sipp
----- Scenario Screen ----- [1-4]: Change Screen --
Call-rate(length)  Port  Total-time  Total-calls  Remote-host
      10 cps(0 ms)  5061      4.01 s       40  127.0.0.1:5060(UDP)

10 new calls during 1.000 s period      16 ms scheduler resolution
0 concurrent calls (limit 30)           Peak was 1 calls, after 0 s
0 out-of-call msg (discarded)
1 open sockets

              Messages  Retrans  Timeout  Unexpected-Msg
INVITE ----->        40         0         0
    100 <-----         0         0         0
    180 <-----         40         0         0
    200 <----- E-RTD    40         0         0
    ACK ----->        40         0
      [    0 ms]
    BYE ----->        40         0         0
    200 <-----         40         0         0

----- [+-|*|/]: Adjust rate ---- [q]: Soft exit ---- [p]: Pause traffic -----

```


2.1 Getting SIPp

SIPp is released under the [GNU GPL license](#). All the terms of the license apply. It was originally created and provided to the SIP community by [Hewlett-Packard](#) engineers in hope it can be useful, but HP does not provide any support nor warranty concerning SIPp.

2.2 SIPp releases

Like many other “open source” projects, there are two versions of SIPp: a stable and unstable release. Stable release: before being labelled as “stable”, a SIPp release is thoroughly tested. So you can be confident that all mentioned features will work :)

Note: Use the stable release for your everyday use and if you are not blocked by a specific feature present in the “unstable release” (see below).

[SIPp stable download page](#)

2.3 Unstable release

Unstable release: all new features and bug fixes are checked in [SIPp’s master tree](#) repository as soon as they are available.

Note: Use the unstable release if you absolutely need a bug fix or a feature that is not in the stable release.

2.4 Available platforms

SIPp is available on Linux and Cygwin. Other Unix distributions are likely to work, but are not tested every release cycle.

Note: SIPp on Cygwin works only on Windows XP and later versions and will not work on Win2000. This is because of IPv6 support.

2.5 Installing SIPp

- On Linux, SIPp is provided in the form of source code. You will need to compile SIPp to actually use it.
- Pre-requisites to compile SIPp are:
 - C++ Compiler
 - curses or ncurses library
 - For TLS support: OpenSSL \geq 0.9.8
 - For pcap play support: libpcap and libnet
 - For SCTP support: lksctp-tools
 - For distributed pauses: [Gnu Scientific Libraries](#)
- You have four options to compile SIPp:
 - Without TLS (Transport Layer Security), SCTP or PCAP support – this is the recommended setup if you don't need to handle SCTP, TLS or PCAP:

```
tar -xvzf sipp-xxx.tar
cd sipp
./configure
make
```

- With TLS support, you must have installed [OpenSSL library](#) (\geq 0.9.8) (which may come with your system). Building SIPp consists only of adding the `--with-openssl` option to the configure command:

```
tar -xvzf sipp-xxx.tar.gz
cd sipp
./configure --with-openssl
make
```

- With PCAP play support:

```
tar -xvzf sipp-xxx.tar.gz
cd sipp
./configure --with-pcap
make
```

- With SCTP support:

```
tar -xvzf sipp-xxx.tar.gz
cd sipp
```

(continues on next page)

(continued from previous page)

```
./configure --with-sctp  
make
```

- You can also combine these various options, e.g.:

```
tar -xvzf sipp-xxx.tar.gz  
cd sipp  
./configure --with-sctp --with-pcap --with-openssl  
make
```

Warning: SIPp compiles under CYGWIN on Windows, provided that you installed IPv6 extension for [CYGWIN](#), as well as libncurses and (optionally OpenSSL and WinPcap). SCTP is not currently supported.

- To compile SIPp on Windows with pcap (media support), you must:
 - Copy the [WinPcap developer package](#) to “C:\cygwin\lib\WpdPack”
 - Remove or rename “pthread.h” in “C:\cygwin\lib\WpdPack\include”, as it interferes with pthread.h from cygwin
 - Compile according to the instructions above.

CHAPTER 3

Main features

SIPp allows to generate one or many SIP calls to one remote system. The tool is started from the command line. In this example, two SIPp are started in front of each other to demonstrate SIPp capabilities.

Run sipp with embedded server (uas) scenario:

```
# ./sipp -sn uas
```

On the same host, run sipp with embedded client (uac) scenario:

```
# ./sipp -sn uac 127.0.0.1
```

3.1 Running SIPp in background

SIPp can be launched in background mode (-bg command line option).

By doing so, SIPp will be detached from the current terminal and run in the background. The PID of the SIPp process is provided. If you didn't specify a number of calls to execute with the -m option, SIPp will run forever.

There is a mechanism implemented to stop SIPp smoothly. The command `kill -SIGUSR1 [SIPp_PID]` will instruct SIPp to stop placing any new calls and finish all ongoing calls before exiting.

When using the background mode, the main sipp instance stops and a child process will continue the job. Therefore, the log files names will contain another PID than the actual sipp instance PID.

3.2 Screens

Several screens are available to monitor SIP traffic. You can change the screen view by pressing 1 to 9 keys on the keyboard.

- Key '1': Scenario screen. It displays a call flow of the scenario as well as some important informations.

```

ocadmin@vista:~/sipp.2004-07-05

----- Scenario Screen ----- [1-4]: Change Screen --
Call-rate(length)  Port    Total-time  Total-calls  Remote-host
      190 cps(0 ms)  5061      50.01 s      8586  127.0.0.1:5060(UDP)

190 new calls during 1.000 s period      3 ms scheduler resolution
205 concurrent calls (limit 570)          Peak was 232 calls, after 6 s
0 out-of-call msg (discarded)
1 open sockets

      Messages  Retrans  Timeout  Unexpected-Msg
INVITE ----->      8586      0      0
    100 <-----      0      0      0
    180 <-----      8586      0      0
    200 <----- B-RTD  8586     68      0
    ACK ----->      8586     68
      [ 1000 ms]
    BYE ----->      8381      0      0
    200 <----- E-RTD  8381      0      0

----- [+-|*|/]: Adjust rate ---- [q]: Soft exit ---- [p]: Pause traffic -----

```

- Key ‘2’: Statistics screen. It displays the main statistics counters. The “Cumulative” column gather all statistics, since SIPp has been launched. The “Periodic” column gives the statistic value for the period considered (specified by -f frequency command line parameter).

```

ocadmin@vista:~/sipp.2004-07-05

----- Statistics Screen ----- [1-4]: Change Screen --
Start Time          | 2004-07-13 17:24:08
Last Reset Time     | 2004-07-13 17:26:05
Current Time        | 2004-07-13 17:26:06

-----+-----+-----
Counter Name        | Periodic value      | Cumulative value
-----+-----+-----
Elapsed Time        | 00:00:00:999        | 00:01:58:019
Call Rate           | 26.026 cps          | 24.886 cps
-----+-----+-----
Incoming call created | 0                   | 0
OutGoing call created | 26                  | 2937
Total Call created   |                      | 2937
Current Call         | 0                   |
-----+-----+-----
Successful call      | 26                  | 2937
Failed call          | 0                   | 0
-----+-----+-----
Response Time        | 00:00:00:000        | 00:00:00:000
Call Length          | 00:00:00:000        | 00:00:00:000
----- [+-|*|/]: Adjust rate ---- [q]: Soft exit ---- [p]: Pause traffic -----

```

- Key '3': Repartition screen. It displays the distribution of response time and call length, as specified in the scenario.

```

ocadmin@vista:~/sipp.2004-07-05

----- Repartition Screen ----- [1-4]: Change Screen --
Average Response Time Repartition
  0 ms <= n < 1000 ms : 0
 1000 ms <= n < 1040 ms : 385
 1040 ms <= n < 1080 ms : 388
 1080 ms <= n < 1120 ms : 384
 1120 ms <= n < 1160 ms : 382
 1160 ms <= n < 1200 ms : 382
      n >= 1200 ms : 190
Average Call Length Repartition
  0 ms <= n < 1000 ms : 0
 1000 ms <= n < 1100 ms : 946
 1100 ms <= n < 1200 ms : 975
 1200 ms <= n < 1300 ms : 190
 1300 ms <= n < 1400 ms : 0
      n >= 1400 ms : 0
----- [+-|*|/]: Adjust rate ---- [q]: Soft exit ---- [p]: Pause traffic -----

```

- Key '4': Variables screen. It displays informations on actions in scenario as well as scenario variable informations.

```

ocadmin@vista:~/sipp.2004-07-05

----- Variables Screen ----- [1-4]: Change Screen --
Action defined Per Message :
=> Message[3] (Receive Message) - [3] action(s) defined :
--> action[0] = Type[1] - where[Full Msg] - checkIt[1] - varId[1]
--> action[1] = Type[1] - where[Full Msg] - checkIt[1] - varId[2]
--> action[2] = Type[1] - where[Header-Contact:] - checkIt[1] - varId[6]

Setted Variable Liste :
=> Variable[1] : setted regexp[([0-9]{1,3}\.){3}[0-9]{1,3}: [0-9]*]
=> Variable[2] : setted regexp[([0-9]{1,3}\.){3}[0-9]{1,3}: [0-9]*]
=> Variable[6] : setted regexp[.*]
----- [+-|*|/]: Adjust rate ---- [q]: Soft exit ---- [p]: Pause traffic -----

```

3.3 Exit codes

To ease automation of testing, upon exit (on fatal error or when the number of asked calls (-m command line option) is reached, sipp exits with one of the following exit codes:

Code	Description
0	All calls were successful
1	At least one call failed
97	Exit on internal command. Calls may have been processed. Also exit on global timeout (see -timeout_global option)
99	Normal exit without calls processed
-1	Fatal error
-2	Fatal error binding a socket

Depending on the system that SIPp is running on, you can echo this exit code by using “echo ?” command.

3.4 Contributing to SIPp

Of course, we welcome contributions, and many of SIPp’s features (including epoll support for better Linux performance, RTP streaming, and Cygwin support) have come from external contributions.

See [developers guide](#) for how to get started

- Richard GAYRAUD [initial code]
- Olivier JACQUES [code/documentation]
- Robert Day [code/documentation]
- Charles P. Wright [code]
- Many contributors [code]

Integrated scenarios

Integrated scenarios? Yes, there are scenarios that are embedded in SIPp executable. While you can create your own custom SIP scenarios (see how to create your own XML scenarios), a few basic (yet useful) scenarios are available in SIPp executable.

4.1 UAC

Scenario file: `uac.xml`

```
SIPp UAC Remote
| (1) INVITE |
|----->|
| (2) 100 (optional) |
|<-----|
| (3) 180 (optional) |
|<-----|
| (4) 200 |
|<-----|
| (5) ACK |
|----->|
|
| (6) PAUSE |
|
| (7) BYE |
|----->|
| (8) 200 |
|<-----|
```

4.2 UAC with media

Scenario file: `uac_pcap.xml`

SIPp	UAC	Remote
	(1) INVITE	
	----->	
	(2) 100 (optional)	
	<-----	
	(3) 180 (optional)	
	<-----	
	(4) 200	
	<-----	
	(5) ACK	
	----->	
	(6) RTP send (8s)	
	=====>	
	(7) RFC2833 DIGIT 1	
	=====>	
	(8) BYE	
	----->	
	(9) 200	
	<-----	

4.3 UAS

Scenario file: `uas.xml`

Remote	SIPp	UAS
	(1) INVITE	
	----->	
	(2) 180	
	<-----	
	(3) 200	
	<-----	
	(4) ACK	
	----->	
	(5) PAUSE	
	(6) BYE	
	----->	
	(7) 200	
	<-----	

4.4 regexp

Scenario file: `regexp.xml`

This scenario, which behaves as an UAC is explained in greater details in this section.

SIPp	regexp	Remote
	(1) INVITE	
	----->	

(continues on next page)

(continued from previous page)

```

| (2) 100 (optional) |
| <-----|
| (3) 180 (optional) |
| <-----|
| (4) 200             |
| <-----|
| (5) ACK             |
| ----->|
|                     |
| (6) PAUSE           |
|                     |
| (7) BYE             |
| ----->|
| (8) 200             |
| <-----|

```

4.5 branch

Scenario files: `branchc.xml` and `branchs.xml`

Those scenarios, which work against each other (branchc for client side and branchs for server side) are explained in greater details in this section.

```

REGISTER ----->
200 <-----
200 <-----
INVITE ----->
100 <-----
180 <-----
403 <-----
200 <-----
ACK ----->
[ 5000 ms]
BYE ----->
200 <-----

```

4.6 UAC Out-of-call Messages

Scenario file: `ooc_default.xml`

When a SIPp UAC receives an out-of-call request, it instantiates an out-of-call scenario. By default this scenario simply replies with a 200 OK response. This scenario can be overridden by passing the `-oocsf` or `-oocsn` command line options.

SIPp UAC	Remote
(1) .*	
<-----	
(2) 200	
----->	

4.7 3PCC

3PCC stands for 3rd Party Call Control. 3PCC is described in [RFC 3725](#). While this feature was first developed to allow 3PCC like scenarios, it can also be used for every case where you would need one SIPp to talk to several remotes.

In order to keep SIPp simple (remember, it's a test tool!), one SIPp instance can only talk to one remote. Which is an issue in 3PCC call flows, like call flow I (SIPp being a controller):

A	Controller	B
(1) INVITE no SDP		
<-----		
(2) 200 offer1		
----->		
	(3) INVITE offer1	
	----->	
	(4) 200 OK answer1	
	<-----	
	(5) ACK	
	----->	
(6) ACK answer1		
<-----		
(7) RTP		
.....		

Scenario file: 3pcc-A.xml

Scenario file: 3pcc-B.xml

Scenario file: 3pcc-C-A.xml

Scenario file: 3pcc-C-B.xml

The 3PCC feature in SIPp allows to have two SIPp instances launched and synchronised together. If we take the example of call flow I, one SIPp instance will take care of the dialog with remote A (this instance is called 3PCC-C-A for 3PCC-Controller-A-Side) and another SIPp instance will take care of the dialog with remote B (this instance is called 3PCC-C-B for 3PCC-Controller-B-Side).

The 3PCC call flow I will, in reality, look like this (Controller has been divided in two SIPp instances):

A	Controller A	Controller B	B
(1) INVITE no SDP			
<-----			
(2) 200 offer1			
----->			
	sendCmd (offer1)		
	----->		
		recvCmd	
		(3) INVITE offer1	
		----->	
		(4) 200 OK answer1	
		<-----	
		sendCmd	
	(answer1)		
	<-----		
	recvCmd	(5) ACK	
	----->		
(6) ACK answer1			
<-----			

(continues on next page)

(continued from previous page)

(7) RTP		
.....		

As you can see, we need to pass information between both sides of the controller. SDP “offer1” is provided by A in message (2) and needs to be sent to B side in message (3). This mechanism is implemented in the scenarios through the <sendCmd> command. This:

```
<sendCmd>
  <![CDATA[
    Call-ID: [call_id]
    [$1]

  ]]>
</sendCmd>
```

Will send a “command” to the twin SIPp instance. Note that including the Call-ID is mandatory in order to correlate the commands to actual calls. In the same manner, this:

```
<recvCmd>
  <action>
    <ereg regexp="Content-Type:.*"
      search_in="msg"
      assign_to="2"/>
  </action>
</recvCmd>
```

Will receive a “command” from the twin SIPp instance. Using the regular expression mechanism, the content is retrieved and stored in a call variable (\$2 in this case), ready to be reinjected:

```
<send>
  <![CDATA[

    ACK sip:[service]@[remote_ip]:[remote_port] SIP/2.0
    Via: SIP/2.0/[transport] [local_ip]:[local_port]
    From: sipp <sip:sipp@[local_ip]:[local_port]>;tag=[call_number]
    To: sut <sip:[service]@[remote_ip]:[remote_port]>[peer_tag_param]
    Call-ID: [call_id]
    CSeq: 1 ACK
    Contact: sip:sipp@[local_ip]:[local_port]
    Max-Forwards: 70
    Subject: Performance Test
    [$2]

  ]]>
</send>
```

In other words, sendCmd and recvCmd can be seen as synchronization points between two SIPp instances, with the ability to pass parameters between each other.

Another scenario that has been reported to be do-able with the 3PCC feature is the following:

- A calls B. B answers. B and A converse
- B calls C. C answers. C and B converse
- B “REFER”s A to C and asks to replace A-B call with B-C call.
- A accepts. A and C talk. B drops out of the calls.

Create your own XML scenarios

5.1 Create your own XML scenarios

Of course embedded scenarios will not be enough. So it's time to create your own scenarios. A SIPp scenario is written in XML (a DTD that may help you write SIPp scenarios does exist and has been tested with jEdit - this is described in a later section). A scenario will always start with:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<scenario name="Basic Sipstone UAC">
```

And end with:

```
</scenario>
```

Easy, huh? Ok, now let's see what can be put inside. You are not obliged to read the whole table now! Just go in the next section for an example.

There are many common attributes used for flow control and statistics, that can be used for all of the message commands (i.e., `<send>` , `<recv>` , `<nop>` , `<pause>` , `<sendCmd>` and `<recvCmd>`).

5.1.1 List of attributes common to all commands

Attribute(s)	Description	Example
start_rtd	Starts one of the " Response Time Duration" timer. (see statistics section).	<code><send start_rtd="invite"></code> the timer named "invite" will start when the message is sent.
rtd	Stops one of the 5 " Response Time Duration"	<code><send rtd="2"></code> the timer number 2 will stop when the message is sent.
repeat_rtd	Used with a rtd attribute, it allows the corresponding " Response Time Duration" timer to be counted more than once per call (useful for loop call flows).	<code><send rtd="1"repeat_rtd=</code> <code>↪"true"></code> the timer number 1 value will be printed but the timer won't stop.
crlf	Displays an empty line after the arrow for the message in main SIPp screen.	<code><send crlf="true"></code>
next	You can put a "next" in any command element to go to another part of the script when you are done with sending the message. For optional receives, the next is only taken if that message was received. See conditional branching section for more info.	Example to jump to label "12" after sending an ACK: <code><send next="12"></code> <code><![CDATA[</code> <code>ACK</code> <code>↪sip:[service]@[remote_</code> <code>↪ip]:[remote_port] SIP/2.</code> <code>↪0</code> <code>Via: ...</code> <code>From: ...</code> <code>To: ...</code> <code>Call-ID: ...</code> <code>Cseq: ...</code> <code>Contact: ...</code> <code>Max-Forwards: ...</code> <code>Subject: ...</code> <code>Content-Length: 0</code> <code>]]></code> <code></send></code> Example to jump to label "5" when receiving a 403 message: <code><recv response="100"</code> <code>optional="true"></code> <code></recv></code> <code><recv response="180"</code> <code>↪optional="true"></code> <code></recv></code> <code><recv response="403"</code> <code>↪optional="true" next="5</code> <code>↪"></code> <code></recv></code> <code><recv response="200"></code> <code></recv></code>

test	You can put a "test" next to a "next" attribute to indicate that you only want to branch to the label specified with "next" if the variable specified in "test" is set (through regexp for	Example to jump to label "6" after sending an ACK only if variable 19 is set: <code><send next="6" test="4"></code> <code><![CDATA[</code>
------	--	---

5.1.2 List of commands with their attributes

Command	Attribute(s)	Description	Example
<send>	retrans	Used for UDP transport only: it specifies the T1 timer value, as described in SIP RFC 3261 , section 17.1.1.2.	<code><send retrans="500 ↪"></code> will initiate T1 timer to 500 milliseconds (RFC 3261 default).
	lost	Emulate packet lost. The value is specified as a percentage.	<code><send lost="10"></code> 10% of the message sent are actually not sent :).
	start_txn	Records the branch ID of this sent message so that responses can be properly matched (without this element the transaction matching is done based on the CSeq method, which is imprecise).	<code><send start_txn= ↪"invite"></code> Stores the branch ID of this message in the transaction named “invite”.
	ack_txn	Indicates that the ACK being sent corresponds to the transaction started by a start_txn attribute. Every INVITE with a start_txn tag must have a matching ACK with an ack_txn attribute.	<code><send ack_txn= ↪"invite"></code> References the branch ID of the transaction named “invite”.
<recv>	response	Indicates what SIP message code is expected.	<code><recv response="200 ↪"></code> SIPp will expect a SIP message with code “200”.
	request	Indicates what SIP message request is expected.	<code><recv request="ACK ↪"></code> SIPp will expect an “ACK” SIP message.

Continued on next page

Table 1 – continued from previous page

Command	Attribute(s)	Description	Example
	optional	Indicates if the message to receive is optional. In case of an optional message and if the message is actually received, it is not seen as a unexpected message. When an unexpected message is received, Sipp looks if this message matches an optional message defined in the previous step of the scenario. If optional is set to “global”, Sipp will look every previous steps of the scenario.	<pre><recv response="100" ↳ " optional="true" ↳ "></pre> <p>The 100 SIP message can be received without being considered as “unexpected”.</p>
	ignorestdp	Ignore SDP from received message, when set to true. It will allow you to reject newly negotiated streams while keeping the old media flowing.	<pre><recv request= ↳ "INVITE" ↳ ignorestdp="true"></pre>
	rrs	Record Route Set. if this attribute is set to “true”, then the “Record-Route:” header of the message received is stored and can be recalled using the [routes] keyword.	<pre><recv response="100" ↳ " rrs="true"></pre>
	auth	Authentication. if this attribute is set to “true”, then the “Proxy-Authenticate:” header of the message received is stored and is used to build the [authentication] keyword.	<pre><recv response="407" ↳ " auth="true"></pre>
	lost	Emulate packet lost. The value is specified as a percentage.	<pre><recv lost="10"></pre> <p>10% of the message received are thrown away.</p>
	timeout	Specify a timeout while waiting for a message. If the message is not received, the call is aborted, unless an ontimeout label is defined.	<pre><recv timeout= ↳ "100000"></pre>

Continued on next page

Table 1 – continued from previous page

Command	Attribute(s)	Description	Example
	ontimeout	Specify a label to jump to if the timeout popped before the message to be received.	Example to jump to label “5” when not receiving a 100 message after 100 seconds: <pre><recv response="100 ↳ " timeout="100000 ↳ " ontimeout="5"> </recv></pre>
	action	Specify an action when receiving the message. See Actions section for possible actions.	Example of a “regular expression” action: <pre><recv response="200 ↳ "> <action> <ereg regexp= ↳ " ([0-9]{1,3}\.) ↳ {3} [0-9]{1,3} : [0- ↳ 9] * " search_in="msg" check_it="true" assign_to="1,2 ↳ "/> </action> </recv></pre>
	regexp_match	Boolean. Indicates if ‘request’ (‘response’ is not available) is given as a regular expression. If so, the recv command will match against the regular expression. This allows to catch several cases in the same receive command.	Example of a recv command that matches MESSAGE or PUBLISH or SUBSCRIBE requests: <pre><recv request= ↳ "MESSAGE PUBLISH SUBSCRIBE ↳ " crlf="true" ↳ regexp_match= ↳ "true"> </recv></pre>
	response_txn	Indicates that this is a response to a transaction that was previously started. To match, the branch ID of the first via header must match the stored transaction ID.	<pre><recv response="200 ↳ " response_txn= ↳ "invite" /></pre> Matches only responses to the message sent with start_txn="invite" attribute.
<pause>	milliseconds	Specify the pause delay, in milliseconds. When this delay is not set, the value of the -d command line parameter is used.	<pre><pause ↳ milliseconds= ↳ "5000"/></pre> pause the scenario for 5 seconds.

Continued on next page

Table 1 – continued from previous page

Command	Attribute(s)	Description	Example
	variable	Indicates which call variable to use to determine the length of the pause.	<pre><pause variable="1 ↪" /></pre> <p>pauses for the number of milliseconds specified by call variable 1.</p>

Continued on next page

Table 1 – continued from previous page

Command	Attribute(s)	Description	Example
	distribution	Indicates which statistical distribution to use to determine the length of the pause. Without GSL, you may use uniform or fixed. With GSL, normal, exponential, gamma, lambda, lognormal, negbin, (negative binomial), pareto, and weibull are available. Depending on the distribution you select, you must also supply distribution specific parameters.	<p>The following examples show the various types of distributed pauses:</p> <ul style="list-style-type: none"> <code><pause distribution="fixed" value="1000" /></code> pauses for 1 second. <code><pause distribution="uniform" min="2000" max="5000"/></code> pauses between 2 and 5 seconds. <p>The remaining distributions require GSL. In general The parameter names were chosen to be as consistent with Wikipedia's distribution description pages.</p> <ul style="list-style-type: none"> <code><pause distribution="normal" mean="60000" stdev="15000"/></code> provides a normal pause with a mean of 60 seconds (i.e. 60,000 ms) and a standard deviation of 15 seconds. The mean and standard deviation are specified as integer milliseconds. The distribution will look like: <code><pause distribution="lognormal" mean="12.28" stdev="1" /></code> creates a distribution's whose natural logarithm has a mean of 12.28 and a standard deviation of 1. The mean and standard deviation are specified as double values (in milliseconds). The distribution will look like:
24		Chapter 5. Create your own XML scenarios	<ul style="list-style-type: none"> <code><pause distribution="exponential" mean="900000"/></code> creates an exponential distribution.

Table 1 – continued from previous page

Command	Attribute(s)	Description	Example
	sanity_check	By default, statistically distributed pauses are sanity checked to ensure that their 99th percentile values are less than INT_MAX. Setting sanity_check to false disables this behavior.	<pre><pause_ ↳distribution= ↳"lognormal" mean= ↳"10" stdev="10" ↳sanity_check= ↳"false"/></pre> <p>disables sanity checking of the lognormal distribution.</p>
<nop>	action	The nop command doesn't do anything at SIP level. It is only there to specify an action to execute. See Actions section for possible actions.	<p>Execute the play_pcap_audio/video action:</p> <pre><nop> <action> <exec play_ ↳pcap_audio="pcap/ ↳g711a.pcap"/> </action> </nop></pre>
<sendCmd>	<![CDATA[]]>	Content to be sent to the twin 3PCC SIPp instance. The Call-ID must be included in the CDATA. In 3pcc extended mode, the From must be included to.	<pre><sendCmd> <![CDATA[Call-ID: [call_ ↳id] [\$1]]]> </sendCmd></pre>
	dest	3pcc extended mode only: the twin sipp instance which the command will be sent to	<pre><sendCmd dest="s1"></pre> <p>the command will be sent to the “s1” twin instance</p>
<recvCmd>	action	Specify an action when receiving the command. See Actions section for possible actions.	<p>Example of a “regular expression” to retrieve what has been send by a send-Cmd command:</p> <pre><recvCmd> <action> <ereg regexp= ↳"Content-Type:.*" ↳search_ ↳in="msg" ↳assign_ ↳to="2"/> </action> </recvCmd></pre>

Continued on next page

Table 1 – continued from previous page

Command	Attribute(s)	Description	Example
	src	3pcc extended mode only: indicate the twin sipp instance which the command is expected to be received from	<pre><recvCmd src = "s1" /></pre> <p>the command will be expected to be received from the “s1” twin instance</p>
<label>	id	A label is used when you want to branch to specific parts in your scenarios. The “id” attribute is an integer where the maximum value is 19. See conditional branching section for more info.	<p>Example: set label number 13:</p> <pre><label id="13" /></pre>
<Response Time Repartition>	value	Specify the intervals, in milliseconds, used to distribute the values of response times.	<pre><ResponseTimeRepartition value="10, 20, 30" /></pre> <p>response time values are distributed between 0 and 10ms, 10 and 20ms, 20 and 30ms, 30 and beyond.</p>
<Call Length Repartition>	value	Specify the intervals, in milliseconds, used to distribute the values of the call length measures.	<pre><CallLengthRepartition value="10, 20, 30" /></pre> <p>call length values are distributed between 0 and 10ms, 10 and 20ms, 20 and 30ms, 30 and beyond.</p>
<Globals>	variables	Specify the name of globally scoped variables.	<pre><Globals variables="foo,bar" /></pre>
<User>	variables	Specify the name of user-scoped variables.	<pre><User variables="foo,bar" /></pre>
<Reference>	variables	Suppresses warnings about unused variables.	<pre><Reference variables="dummy" /></pre>

There are not so many commands: send, recv, sendCmd, recvCmd, pause, ResponseTimeRepartition, CallLengthRepartition, Globals, User, and Reference. To make things even clearer, nothing is better than an example...

5.1.3 Structure of client (UAC like) XML scenarios

A client scenario is a scenario that starts with a “send” command. So let’s start:

```
<scenario name="Basic Sipstone UAC">
  <send>
    <![CDATA[

      INVITE sip:[service]@[remote_ip]:[remote_port] SIP/2.0
      Via: SIP/2.0/[transport] [local_ip]:[local_port]
      From: sipp <sip:sipp@[local_ip]:[local_port]>;tag=[call_number]
      To: sut <sip:[service]@[remote_ip]:[remote_port]>
      Call-ID: [call_id]
      Cseq: 1 INVITE
      Contact: sip:sipp@[local_ip]:[local_port]
      Max-Forwards: 70
      Subject: Performance Test
      Content-Type: application/sdp
      Content-Length: [len]

      v=0
      o=user1 53655765 2353687637 IN IP[local_ip_type] [local_ip]
      s=-
      t=0 0
      c=IN IP[media_ip_type] [media_ip]
      m=audio [media_port] RTP/AVP 0
      a=rtpmap:0 PCMU/8000

    ]]>
  </send>
```

Inside the “send” command, you have to enclose your SIP message between the “<![CDATA” and the “]]>” tags. Everything between those tags is going to be sent toward the remote system. You may have noticed that there are strange keywords in the SIP message, like [service], [remote_ip], Those keywords are used to indicate to SIPp that it has to do something with it.

Now that the INVITE message is sent, SIPp can wait for an answer by using the “recv” command.

```
<recv response="100"> optional="true"
</recv>

<recv response="180"> optional="true"
</recv>

<recv response="200">
</recv>
```

100 and 180 messages are optional, and 200 is mandatory. In a “recv” sequence, there must be one mandatory message .

Now, let’s send the ACK:

```
<send>
  <![CDATA[

    ACK sip:[service]@[remote_ip]:[remote_port] SIP/2.0
    Via: SIP/2.0/[transport] [local_ip]:[local_port]
```

(continues on next page)

(continued from previous page)

```

From: sipp <sip:sipp@[local_ip]:[local_port]>;tag=[call_number]
To: sut <sip:[service]@[remote_ip]:[remote_port]>[peer_tag_param]
Call-ID: [call_id]
Cseq: 1 ACK
Contact: sip:sipp@[local_ip]:[local_port]
Max-Forwards: 70
Subject: Performance Test
Content-Length: 0

]]>
</send>

```

We can also insert a pause. The scenario will wait for 5 seconds at this point.

```
<pause milliseconds="5000" />
```

And finish the call by sending a BYE and expecting the 200 OK:

```

<send retrans="500">
  <![CDATA[

    BYE sip:[service]@[remote_ip]:[remote_port] SIP/2.0
    Via: SIP/2.0/[transport] [local_ip]:[local_port]
    From: sipp <sip:sipp@[local_ip]:[local_port]>;tag=[call_number]
    To: sut <sip:[service]@[remote_ip]:[remote_port]>[peer_tag_param]
    Call-ID: [call_id]
    Cseq: 2 BYE
    Contact: sip:sipp@[local_ip]:[local_port]
    Max-Forwards: 70
    Subject: Performance Test
    Content-Length: 0

  ]]>
</send>

<recv response="200" />

```

And this is the end of the scenario:

```
</scenario>
```

Creating your own SIPp scenarios is not a big deal. If you want to see other examples, use the `-sd` parameter on the command line to display embedded scenarios.

5.1.4 Structure of server (UAS like) XML scenarios

A server scenario is a scenario that starts with a “recv” command. The syntax and the list of available commands is the same as for “client” scenarios.

But you are more likely to use `[last_*]` keywords in those server side scenarios. For example, a UAS example will look like:

```

<recv request="INVITE">
</recv>

```

(continues on next page)

(continued from previous page)

```

<send>
  <![CDATA[

    SIP/2.0 180 Ringing
    [last_Via:]
    [last_From:]
    [last_To:];tag=[call_number]
    [last_Call-ID:]
    [last_CSeq:]
    Contact: <sip:[local_ip]:[local_port];transport=[transport]>
    Content-Length: 0

  ]]>
</send>

```

The answering message, 180 Ringing in this case, is built with the content of headers received in the INVITE message.

5.2 Keyword list

If you want a literal left bracket instead of a keyword, use `\x5b` where you want a `[`.

5.2.1 [service]

Default (service)

Description Service field, as passed in the `-s service_name`

5.2.2 [remote_ip]

Description Remote IP address, as passed on the command line.

5.2.3 [remote_port]

Default 5060

Description Remote IP port, as passed on the command line. You can add a computed offset `[remote_port+3]` to this value.

5.2.4 [transport]

Default UDP

Description Depending on the value of `-t` parameter, this will take the values “UDP” or “TCP”.

5.2.5 [local_ip]

Description (Primary host IP address) Will take the value of `-i` parameter.

5.2.6 [local_ip_type]

Description Depending on the address type of -i parameter (IPv4 or IPv6), local_ip_type will have value “4” for IPv4 and “6” for IPv6.

5.2.7 [local_port]

Default Chosen by the system

Description Will take the value of -p parameter. You can add a computed offset [local_port+3] to this value.

5.2.8 [len]

Description Computed length of the SIP body. To be used in “Content-Length” header. You can add a computed offset [len+3] to this value.

5.2.9 [call_number]

Description Index. The call_number starts from “1” and is incremented by 1 for each call.

5.2.10 [cseq]

Description Generates automatically the CSeq number. The initial value is 1 by default. It can be changed by using the -base_cseq command line option.

5.2.11 [call_id]

Description A call_id identifies a call and is generated by SIPp for each new call. In client mode, it is mandatory to use the value generated by SIPp in the “Call-ID” header. Otherwise, SIPp will not recognise the answer to the message sent as being part of an existing call. Note: [call_id] can be pre-pended with an arbitrary string using ‘///’. Example: Call-ID: ABCDEFGHIJ///[call_id] - it will still be recognized by SIPp as part of the same call.

5.2.12 [media_ip]

Description Depending on the value of -mi parameter, it is the local IP address for RTP echo.

5.2.13 [media_ip_type]

Description Depending on the address type of -mi parameter (IPv4 or IPv6), media_ip_type will have value “4” for IPv4 and “6” for IPv6. Useful to build the SDP independently of the media IP type.

5.2.14 [media_port]

Description Depending on the value of -mp parameter, it set the local RTP echo port number. Default is none. RTP/UDP packets received on that port are echoed to their sender. You can add a computed offset [media_port+3] to this value.

5.2.15 [auto_media_port]

Description Only for pcap. To make audio and video ports begin from the value of -mp parameter, and change for each call using a periodical system, modulo 10000 (which limits to 10000 concurrent RTP sessions for pcap_play)

5.2.16 [last_*]

Description The '[last_*]' keyword is replaced automatically by the specified header if it was present in the last message received (except if it was a retransmission). If the header was not present or if no message has been received, the '[last_*]' keyword is discarded, and all bytes until the end of the line are also discarded. If the specified header was present several times in the message, all occurrences are concatenated (CRLF separated) to be used in place of the '[last_*]' keyword.

5.2.17 [field0-n file=<filename> line=<number>]

Description Used to inject values from an external CSV file. See “Injecting values from an external CSV during calls” section. The optional file and line parameters allow you to select which of the injection files specified on the command line to use and which line number from that file.

5.2.18 [file name=<filename>]

Description Inserts the entire contents of filename into the message. Whitespace, including carriage returns and newlines at the end of the line in the file are not processed as with other keywords; thus your file must be formatted exactly as you would like the bytes to appear in the message.

5.2.19 [timestamp]

Description The current time using the same format as error log messages.

5.2.20 [last_message]

Description The last received message.

5.2.21 [\$n]

Description Used to inject the value of call variable number n. See *Actions_* section

5.2.22 [authentication]

Description Used to put the authentication header. This field can have parameters, in the following form: [authentication username=myusername password=mypassword]. If no username is provided, the value from the -au (authentication username) or -s (service) command line parameter is used. If no password is provided, the value from -ap command line parameter is used. See “Authentication” section

5.2.23 [pid]

Description Provide the process ID (pid) of the main SIPp thread.

5.2.24 [routes]

Description If the “rrs” attribute in a recv command is set to “true”, then the “Record-Route:” header of the message received is stored and can be recalled using the [routes] keyword.

5.2.25 [next_url]

Description If the “rrs” attribute in a recv command is set to “true”, then the [next_url] contains the contents of the Contact header (i.e within the ‘<’ and ‘>’ of Contact)

5.2.26 [branch]

Description Provide a branch value which is a concatenation of magic cookie (z9hG4bK) + call number + message index in scenario. An offset (like [branch-N]) can be appended if you need to have the same branch value as a previous message.

5.2.27 [msg_index]

Description Provide the message number in the scenario.

5.2.28 [cseq]

Description Provides the CSeq value of the last request received. This value can be incremented (e.g. [cseq+1] adds 1 to the CSeq value of the last request).

5.2.29 [clock_tick]

Description Includes the internal SIPp clock tick value in the message.

5.2.30 [sipp_version]

Description Includes the SIPp version string in the message.

5.2.31 [tdmmap]

Description Includes the tdm map values used by the call in the message (see -tdmmap option).

5.2.32 [fill]

Description Injects filler characters into the message. The length of the fill text is equal to the call variable stored in the variable=N parameter. By default the text is a sequence of X's, but can be controlled with the text=”text” parameter.

5.2.33 [users]

Description If the `-users` command line option is specified, then this keyword contains the number of users that are currently instantiated.

5.2.34 [userid]

Description If the `-users` command line option is specified, then this keyword contains the integer identifier of the current user (starting at zero and ending at `[users-1]`).

5.3 Actions

In a `recv` or `recvCmd` command, you have the possibility to execute an action. Several actions are available:

- *Regular expressions* (ereg)
- Log something in aa log file (log)
- Execute an external (system), internal (int_cmd) or pcap_play_audio/pcap_play_video command (exec)
- Manipulate double precision variables using arithmetic
- Assign string values to a variable
- Compare double precision variables
- Jump to a particular scenario index
- Store the current time into variables
- Lookup a key in an indexed injection file
- Verify Authorization credentials
- Change a Call's Network Destination

5.3.1 Regular expressions

Using regular expressions in SIPp allows to

- Extract content of a SIP message or a SIP header and store it for future usage (called re-injection)
- Check that a part of a SIP message or of an header is matching an expected expression

Regular expressions used in SIPp are defined per 'Posix Extended standard (POSIX 1003.2)'. If you want to learn how to write regular expressions, I will recommend this 'regex tutorial'.

Here is the syntax of the regexp action:

regex action syntax

Key-word	De-fault	Description
reg-exp	None	Contains the regex to use for matching the received message or header. MANDATORY.
search_in	msg	can have four values: “msg” (try to match against the entire message), “hdr” (try to match against a specific SIP header), “body” (try to match against the SIP message body), or “var” (try to match against a SIPp string variable).
header	None	Header to try to match against. Only used when the search_in tag is set to hdr. MANDATORY IF search_in is equal to hdr.
variable	None	Variable to try to match against. Only used when the search_in tag is set to var. MANDATORY IF search_in is equal to var.
case_sensitive	false	To look for a header ignoring case . Only used when the search_in tag is set to hdr.
occurrence	1	To find the nth occurrence of a header. Only used when the search_in tag is set to hdr.
start_line	false	To look only at start of line. Only used when the search_in tag is set to hdr.
check_it	false	if set to true, the call is marked as failed if the regex doesn't match. Can not be combined with check_it_inverse.
check_it_inverse	false	inverse of check_it. iff set to true, the call is marked as failed if the regex does match. Can not be combined with check_it.
assign_to	None	contain the variable id (integer) or a list of variable id which will be used to store the result(s) of the matching process between the regex and the message. Those variables can be re-used at a later time either by using ‘[\$n]’ in the scenario to inject the value of the variable in the messages or by using the content of the variables for conditional branching. The first variable in the variable list of assign_to contains the entire regular expression matching. The following variables contain the sub-expressions matching.

Example for assign_to

```
<ereg regexp="o=([[:alnum:]]*) ([[:alnum:]]*) ([[:alnum:]]*)"
  search_in="msg"
  check_it=i"true"
  assign_to="3,4,5,8"/>
```

If the SIP message contains the line

```
o=user1 53655765 2353687637 IN IP4 127.0.0.1
```

variable 3 contains “o=user1 53655765 2353687637”, variable 4 contains “user1”, variable 5 contains “53655765” and variable 8 contains “2353687637”. Note that you can have several regular expressions in one action.

The following example is used to:

- First action:
 - Extract the first IPv4 address of the received SIP message
 - Check that we could actually extract this IP address (otherwise call will be marked as failed)
 - Assign the extracted IP address to call variables 1 and 2.
- Second action:

- Extract the Contact: header of the received SIP message
- Assign the extracted Contact: header to variable 6.

```
<recv response="200" start_rtd="true">
  <action>
    <ereg regexp="([0-9]{1,3}\.){3}[0-9]{1,3}:[0-9]*" search_in="msg" check_it="true"
    ↪assign_to="1,2" />
    <ereg regexp=".*" search_in="hdr" header="Contact:" check_it="true" assign_to="6"
    ↪/>
  </action>
</recv>
```

5.3.2 Log a message

The “log” action allows you to customize your traces. Messages are printed in the <scenario file name>_<pid>_logs.log file. Any keyword is expanded to reflect the value actually used.

Warning: Logs are generated only if -trace_logs option is set on the command line.

Example:

```
<recv request="INVITE" crlf="true" rrs="true">
  <action>
    <ereg regexp=".*" search_in="hdr" header="Some-New-Header:" assign_to="1" />
    <log message="From is [last_From]. Custom header is [$1]" />
  </action>
</recv>
```

You can use the alternative “warning” action to log a message to SIPp’s error log. For example:

```
<warning message="From is [last_From]. Custom header is [$1]" />
```

5.3.3 Execute a command

The “exec” action allows you to execute “internal”, “external”, “play_pcap_audio” or “play_pcap_video” commands.

5.3.4 Internal commands

Internal commands (specified using int_cmd attribute) are stop_call, stop_gracefully (similar to pressing ‘q’), stop_now (similar to ctrl+C).

Example that stops the execution of the script on receiving a 603 response:

```
<recv response="603" optional="true">
  <action>
    <exec int_cmd="stop_now" />
  </action>
</recv>
```

5.3.5 External commands

External commands (specified using command attribute) are anything that can be executed on local host with a shell.

Example that execute a system echo for every INVITE received:

```
<recv request="INVITE">
  <action>
    <exec command="echo [last_From] is the from header received >> from_list.log"/>
  </action>
</recv>
```

5.3.6 Media/RTP commands

RTP streaming allows you to stream audio from a PCMA, PCMU, G722, iLBC or G729-encoded audio file (e.g. a .wav file). The “rtp_stream” action controls this.

- `<exec rtp_stream="file.wav" />` will stream the audio contained in file.wav, assuming it is a PCMA-format file.
- `<exec rtp_stream="[filename],[loopcount],[payloadtype]" />` will stream the audio contained in [filename], repeat the stream [loopcount] times (the default is 1, and -1 indicates it will repeat forever), and will treat the audio as being of [payloadtype] (where 8 is the default of PCMA, 0 indicates PCMU, 9 indicates G722, 18 indicates G729 and 98 indicates iLBC in 30ms 13.33kbps).
- `<exec rtp_stream="pause" />` will pause any currently active playback.
- `<exec rtp_stream="resume" />` will resume any currently paused playback.

PCAP play commands (specified using play_pcap_audio / play_pcap_video attributes) allow you to send a pre-recorded RTP stream using the [pcap library](#). Choose play_pcap_audio to send the pre-recorded RTP stream using the “m=audio” SIP/SDP line port as a base for the replay.

Choose play_pcap_video to send the pre-recorded RTP stream using the “m=video” SIP/SDP line port as a base.

The play_pcap_audio/video command has the following format: play_pcap_audio="[file_to_play]" with:

- file_to_play: the pre-recorded pcap file to play

The audio file should be the raw samples, example files are included for PCMA, G722 and iLBC (mode=30).

Codec	Payload id	Packet size	Packet time	FFmpeg arguments
PCMU	0	160 bytes	20 ms	-f ulaw -ar 8k -ac 1
PCMA	8	160 bytes	20 ms	-f alaw -ar 8k -ac 1
G722	9	160 bytes	20 ms	-f g722 -ar 16k -ac 1
G729	18	20 bytes	20 ms	<i>not supported by ffmpeg</i>
iLBC	98	50 bytes	30 ms	-f ilbc -ar 8k -ac 1 -b:a 13.33k

Note: Ffmpeg adds a header to iLBC files denoting the mode that is used, either 20 or 30 ms per packet. This header needs to be stripped from the file.

Note: The action is non-blocking. SIPp will start a light-weight thread to play the file and the scenario with continue immediately. If needed, you will need to add a pause to wait for the end of the pcap play.

Warning: A known bug means that starting a `pcap_play_audio` command will end any `pcap_play_video` command, and vice versa; you cannot play both audio and video streams at once.

Example that plays a pre-recorded RTP stream:

```
<nop>
  <action>
    <exec play_pcap_audio="pcap/g711a.pcap" />
  </action>
</nop>
```

5.3.7 Variable Manipulation

You may also perform simple arithmetic (add, subtract, multiply, divide) on floating point values. The “assign_to” attribute contains the first operand, and is also the destination of the resulting value. The second operand is either an immediate value or stored in a variable, represented by the “value” and “variable” attributes, respectively.

SIPp supports call variables that take on double-precision floating values. The actions that modify double variables all write to the variable referenced by the `assign_to` parameter. These variables can be assigned using one of three actions: `assign`, `sample`, or `todouble`. For `assign`, the double precision value is stored in the “value” parameter. The `sample` action assigns values based on statistical distributions, and uses the same parameters as a statistically distributed pauses. Finally, the `todouble` command converts the variable referenced by the “variable” attribute to a double before assigning it.

For example, to assign the value 1.0 to \$1 and sample from the normal distribution into \$2:

```
<nop>
  <action>
    <assign assign_to="1" value="1" />
    <sample assign_to="2" distribution="normal" mean="0" stdev="1"/>
    <!-- Stores the first field in the injection file into string variable $3.
         You may also use regular expressions to store string variables. -->
    <assignstr assign_to="3" value="[field0]" />
    <!-- Converts the string value in $3 to a double-precision value stored in $4. -->
    <todouble assign_to="4" variable="3" />
  </action>
</nop>
```

Simple arithmetic is also possible using the `<add>`, `<subtract>`, `<multiply>`, and `<divide>` actions, which add, subtract, multiply, and divide the variable referenced by `assign_to` by the value in `value`. For example, the following action modifies variable one as follows:

```
<nop>
  <action>
    <assign assign_to="1" value="0" /> <!-- $1 == 0 -->
    <add assign_to="1" value="2" /> <!-- $1 == 2 -->
    <subtract assign_to="1" value="3" /> <!-- $1 == -1 -->
    <multiply assign_to="1" value="4" /> <!-- $1 == -4 -->
    <divide assign_to="1" value="5" /> <!-- $1 == -0.8 -->
  </action>
</nop>
```

Rather than using fixed values, you may also retrieve the second operand from a variable, using the `<variable>` parameter. For example:

```
<nop>
  <action>
    <!-- Multiplies $1 by itself -->
    <multiply assign_to="1" variable="1" />
    <!-- Divides $1 by $2, Note that $2 must not be zero -->
    <multiply assign_to="1" variable="2" />
  </action>
</nop>
```

5.3.8 String Variables

You can create string variables by using the `<assignstr>` command, which accepts two parameters: `assign_to` and `value`. The value may contain any of the same substitutions that a message can contain. For example:

```
<nop>
  <action>
    <!-- Assign the value in field0 of the CSV file to a $1. -->
    <assignstr assign_to="1" value="[field0]" />
  </action>
</nop>
```

A string variable and a value can be compared using the `<strcmp>` action. The result is a double value, that is less than, equal to, or greater than zero if the variable is lexicographically less than, equal to, or greater than the value. The parameters are `assign_to`, `variable`, and `value`. For example:

```
<nop>
  <action>
    <!-- Compare the value of $strvar to "Hello" and assign it to $result.. -->
    <strcmp assign_to="result" variable="strvar" value="Hello" />
  </action>
</nop>
```

5.3.9 Variable Testing

Variable testing allows you to construct loops and control structures using call variables. The test action takes four arguments: `variable` which is the variable that to compare against `value`, and `assign_to` which is a boolean call variable that the result of the test is stored in. Compare may be one of the following tests: `equal`, `not_equal`, `greater_than`, `less_than`, `greater_than_equal`, or `less_than_equal`.

Example that sets `$2` to true if `$1` is less than 10:

```
<nop>
  <action>
    <test assign_to="2" variable="1" compare="less_than" value="10" />
  </action>
</nop>
```

5.3.10 lookup

The lookup action is used for indexed injection files (see indexed injection files). The lookup action takes a file and key as input and produces an integer line number as output. For example the following action extracts the username from an authorization header and uses it to find the corresponding line in `users.csv`.

```
<recv request="REGISTER">
  <action>
    <ereg regexp="Digest .*username=\"([^\"]*)\" search_in="hdr" header=
→ "Authorization:" assign_to="junk,username" />
    <lookup assign_to="line" file="users.csv" key="[$username]" />
  </action>
</recv>
```

5.3.11 Updating In-Memory Injection files

Injection files, particularly when an index is defined can serve as an in-memory data store for your SIPp scenario. The `<insert>` and `<replace>` actions provide a method of programmatically updating SIPp's in-memory version of an injection file (there is presently no way to update the disk-based version). The insert action takes two parameters: file and value, and the replace action takes an additional line value. For example, to inserting a new line can be accomplished as follows:

```
<nop display="Insert User">
  <action>
    <insert file="usersdb.conf" value="[$user];[$calltype]" />
  </action>
</nop>
```

Replacing a line is similar, but a line number must be specified. You will probably want to use the lookup action to obtain the line number for use with replace as follows:

```
<nop display="Update User">
  <action>
    <lookup assign_to="index" file="usersdb.conf" key="[$user]" />
    <!-- Note: This assumes that the lookup always succeeds. -->
    <replace file="usersdb.conf" line="[$index]" value="[$user];[$calltype]" />
  </action>
</nop>
```

5.3.12 Jumping to an Index

You can jump to an arbitrary scenario index using the `<jump>` action. This can be used to create rudimentary subroutines. The caller can save their index using the `[msg_index]` substitution, and the callee can jump back to the same place using this action. If there is a special label named `“_unexp.main”` in the scenario, SIPp will jump to that label whenever an unexpected message is received and store the previous address in the variable named `“_unexp.retaddr”`.

Example that jumps to index 5:

```
<nop>
  <action>
    <jump value="5" />
  </action>
</nop>
```

Example that jumps to the index contained in the variable named `_unexp.retaddr`:

```
<nop>
  <action>
    <jump variable="_unexp.retaddr" />
```

(continues on next page)

(continued from previous page)

```

</action>
</nop>

```

5.3.13 gettimeofday

The gettimeofday action allows you to get the current time in seconds and microseconds since the epoch. For example:

```

<nop>
  <action>
    <gettimeofday assign_to="seconds,microseconds" />
  </action>
</nop>

```

5.3.14 setdest

The setdest action allows you to change the remote end point for a call. The parameters are the transport, host, and port to connect the call to. There are certain limitations based on SIPp's design: you can not change the transport for a call; and if you are using TCP then multi-socket support must be selected (i.e. -t tn must be specified). Also, be aware that frequently using setdest may reduce SIPp's capacity as name resolution is a blocking operation (thus potentially causing SIPp to stall while looking up host names). This example connects to the value specified in the [next_url] keyword.

```

<nop>
  <action>
    <assignstr assign_to="url" value="[next_url]" />
    <ereg regexp="sip:.*@([0-9A-Za-z\.]+) : ([0-9]+);transport=([A-Z]+)" search_in="var
→ " check_it="true" assign_to="dummy,host,port,transport" variable="url" />
    <setdest host="[$host]" port="[$port]" protocol="[$transport]" />
  </action>
</nop>

```

Warning: If you are using setdest with IPv6, you must not use square brackets around the address. These have a special meaning to SIPp, and it will try to interpret your IPv6 address as a variable. Since the port is specified separately, square brackets are never necessary.

5.3.15 verifyauth

The verifyauth action checks the Authorization header in an incoming message against a provided username and password. The result of the check is stored in a boolean variable. This allows you to simulate a server which requires authorization. Currently only simple MD5 digest authentication is supported. Before using the verifyauth action, you must send a challenge. For example:

```

<recv request="REGISTER" />
<send>
  <![CDATA[

    SIP/2.0 401 Authorization Required
    [last_Via:]
    [last_From:]

```

(continues on next page)

(continued from previous page)

```

    [last_To:];tag=[pid]SIPpTag01[call_number]
    [last_Call-ID:]
    [last_CSeq:]
    Contact: <sip:[local_ip]:[local_port];transport=[transport]>
    WWW-Authenticate: Digest realm="test.example.com", nonce=
    ↪ "47ebe028cd119c35d4877b383027d28da013815"
    Content-Length: [len]

  ]]>
</send>

```

After receiving the second request, you can extract the username provided and compare it against a list of user names and passwords provided as an injection file, and take the appropriate action based on the result:

```

<recv request="REGISTER">
  <action>
    <ereg regexp="Digest .*username=\"([^\"]*)\" search_in="hdr" header=
    ↪ "Authorization:" assign_to="junk,username" />
    <lookup assign_to="line" file="users.conf" key="[username]" />
    <verifyauth assign_to="authvalid" username="[field0 line=\"[$line]\"]" password=
    ↪ "[field3 line=\"[$line]\"]" />
  </action>
</recv>

<nop hide="true" test="authvalid" next="goodauth" />
<nop hide="true" next="badauth" />

```

5.4 Variables

For complex scenarios, you will need to store bits of information that can be used across messages or even calls. Like other programming languages, SIPp's XML scenario definition allows you to use variables for this purpose. A variable in SIPp is referenced by an alphanumeric name. In past versions of SIPp, variables names were numeric only; thus in this document and the embedded scenarios, you are likely to see lots of variables of the form "1", "2", etc.; although when creating new scenarios you are encouraged to assign meaningful names to your variables.

Aside from a name, SIPp's variables are also loosely typed. The type of a variable is not explicitly declared, but is instead inferred from the action that set it. There are four types of variables: string, regular expression matches, doubles, and booleans. All mathematical operations take place on doubles. The **<test>** and **<verifyauth>** actions create boolean values. String variables and regular expression matches are similar. When a string's value is called for, a regular expression match can be substituted. The primary difference is related to the test attribute (see [Conditional branching](#)). If a string has been defined, a test is evaluated to true. However, for a regular expression variable, the regular expression that set it must match for the test to be evaluated to true. Values can be converted to strings using the **<assignstr>** action. Values can be converted to doubles using the **<todouble>** action.

Variables also have a scope, which is one of global to all calls, per- user, or the default per-call. A global variable can be used, for example to store scenario configuration parameters or to keep a global counter. A user-variable when combined with the -users option allows you to keep per-user state across calls (e.g., if this user has already registered). Finally, the default per-call variables are useful for copying values from one SIP message to the next or controlling branching. Variables can be declared globally or per-user using the following syntax:

```

<Global variables="foo,bar" />
<User variables="baz,quux" />

```

Local variables need not be declared. To prevent programming errors, SIPp performs very rudimentary checks to ensure that each variable is used more than once in the scenario (this helps prevent some typos from turning into hard to debug errors). Unfortunately, this can cause some complication with regular expression matching. The regular expression action must assign the entire matched expression to a variable. If you are only interested in checking the validity of the expression (i.e. the `check_it` attribute is set) or in capturing a sub-expression, you must still assign the entire expression to a variable. As this variable is likely only referenced once, you must inform SIPp that you are knowingly using this variable once with a Reference clause. For example:

```
<recv request="INVITE">
  <action>
    <ereg regexp="<sip: ([^;@]*) " search_in="hdr" header="To:" assign_to="dummy,uri" />
  </action>
</recv>
<Reference variables="dummy" />
```

5.5 Injecting values from an external CSV during calls

You can use “-inf file_name” as a command line parameter to input values into the scenarios. The first line of the file should say whether the data is to be read in sequence (SEQUENTIAL), random order (RANDOM), or in a user based manner (USER). Each line corresponds to one call and has one or more ‘;’ delimited data fields and they can be referred as [field0], [field1], ... in the xml scenario file. Example:

```
SEQUENTIAL
#This line will be ignored
Sarah;sipphone32
Bob;sipphone12
#This line too
Fred;sipphone94
```

Will be read in sequence (first call will use first line, second call second line). At any place where the keyword “[field0]” appears in the scenario file, it will be replaced by either “Sarah”, “Bob” or “Fred” depending on the call. At any place where the keyword “[field1]” appears in the scenario file, it will be replaced by either “sipphone32” or “sipphone12” or “sipphone94” depending on the call. At the end of the file, SIPp will re-start from the beginning. The file is not limited in size.

You can override the default line selection strategy with the optional line argument. For example:

```
[field0 line=1]
```

Selects the second line in the file (the first line is line zero. The line parameters support keywords in the argument, so in conjunction with a lookup action it is possible to select values based on a key.

The CSV file can contain comment lines. A comment line is a line that starts with a “#”.

As a picture says more than 1000 words, here is one:

Think of the possibilities of this feature. They are huge.

It is possible to use more than one injection file, and is necessary when you want to select different types of data in different ways. For example, when running a user-based benchmark, you may have a caller.csv with “USER” as the first line and a callee.csv with “RANDOM” as the first line. To specify which CSV file is used, add the `file=` parameter to the keyword. For example:


```
INVITE sip:[field0 file="callee.csv"] SIP/2.0
From: sipp user <[field0 file="caller.csv"]>;tag=[pid]SIPpTag00[call_number]
To: sut user <[field0 file="callee.csv"]>
...
```

Will select the destination user from callee.csv and the sending user from caller.csv. If no file parameter is specified, then the first input file on the command line is used by default.

5.5.1 PRINTF Injection files

An extension of the standard injection file is a “PRINTF” injection file. Often, an input file will has a repetitive nature such as:

```
USERS
user000;password000
user001;password001
...
user999;password999
```

SIPp must maintain this structure in memory, which can reduce performance for very large injection files. To eliminate this problem, SIPp can automatically generate such a structured file based on one or more template lines. For example:

```
USERS,PRINTF=999
user%03d;password%03d
```

Has the same logical meaning as the original example, yet SIPp only needs to store one entry in memory. Each time a line is used; SIPp will replace %d with the requested line number (starting from zero). Standard printf format decimal specifiers can be used. When more than one template line is available, SIPp cycles through them. This example:

```
USERS,PRINTF=4
user%03d;password%03d;Foo
user%03d;password%03d;Bar
```

Is equivalent to the following injection file:

```
USERS
user000;password000;Foo
user001;password001;Bar
user002;password002;Foo
user003;password003;Bar
```

The following parameters are used to control the behavior of printf injection files:

5.6 Printf Injection File Parameters

Parameter	Description	Example
PRINTF	How many virtual lines exist in this file.	PRINTF=10, creates 10 virtual lines
PRINTFMULTIPLE	Multiple the virtual line number by this value before generating the substitutions used.	PRINTF=10,PRINTFMULTIPLE=2 creates 10 virtual lines numbered 0,2,4,...,18.
PRINTFOFFSET	Add this value to the virtual line number before generating the substitutions used (applied after PRINTFMULTIPLE).	PRINTF=10,PRINTFOFFSET=100 creates 10 virtual lines numbered 100-109.
		PRINTF=10,PRINTFMULTIPLE=2,PRINTFOFFSET=10 creates 10 users numbered 10,12,14,... 28.

5.6.1 Indexing Injection files

The `-inindex` option allows you to generate an index of an injection file. The arguments to `-inindex` are the injection file to index and the field number that should be indexed. For example if you have an injection file that contains user names and passwords (as the following):

```
USERS
alice,pass_A
bob,pass_B
carol,pass_C
```

You may want to extract the password for a given user in the file. To do this efficiently, SIPp must build an index for the first field (0). Thus you would pass the argument `-inindex users.csv 0` (assuming the file basename is `users.csv`). SIPp will create an index that contains the logical entries `{"alice" => 0, "bob" => 1, "carol" => 2}`. To extract a particular password, you can use the lookup action to store the line number into a variable (say `$line`) and then use the keyword `[field1 line="$line"]`.

5.7 Conditional branching

5.7.1 Conditional branching in scenarios

It is possible to execute a scenario in a non-linear way. You can jump from one part of the scenario to another for example when a message is received or if a call variable is set.

You define a label (in the xml) as `<label id="n"/>` Where `n` is a number between 1 and 19 (we can easily have more if needed). The label commands go anywhere in the main scenario between other commands. To any action command (send, receive, pause, etc.) you add a `next="n"` parameter, where `n` matches the id of a label. When it has done the command it continues the scenario from that label. This part is useful with optional receives like 403 messages, because it allows you to go to a different bit of script to reply to it and then rejoin at the BYE (or wherever or not).

Alternatively, if you add a `test="m"` parameter to the next, it goes to the label only if variable `[$m]` is set. This allows you to look for some string in a received packet and alter the flow either on that or a later part of the script. The evaluation of a test varies based on the type of call variable. For regular expressions, at least one match must have been found; for boolean variables the value must be true; and for all others a value must have been set (currently this only applies to doubles). For more complicated tests, see the `<test>` action.

Warning: If you add special cases at the end, dont forget to put a label at the real end and jump to it at the end of the normal flow.

Example:

The following example corresponds to the embedded ‘branchc’ (client side) scenario. It has to run against the embedded ‘branches’ (server side) scenario.

5.7.2 Randomness in conditional branching

To have SIPp behave somewhat more like a “normal” SIP client being used by a human, it is possible to use “statistical branching”. Wherever you can have a conditional branch on a variable being set (`test="4"`), you can also branch based on a statistical decision using the attribute “chance” (e.g. `chance="0.90"`). Chance can have a value between 0 (never)

and 1 (always). “test” and “chance” can be combined, i.e. only branching when the test succeeds and the chance is good.

With this, you can have a variable reaction in a given scenario (e.g.. answer the call or reject with busy), or run around in a loop (e.g. registrations) and break out of it after some random number of iterations.

5.8 SIP authentication

SIPp supports SIP authentication. Two authentication algorithm are supported: Digest/MD5 (“algorithm=”MD5””) and Digest/AKA (“algorithm=”AKAv1-MD5””, as specified by 3GPP for IMS).

Enabling authentication is simple. When receiving a 401 (Unauthorized) or a 407 (Proxy Authentication Required), you must add `auth=true` in the `<recv>` command to take the challenge into account. Then, the authorization header can be re-injected in the next message by using `[authentication]` keyword.

Computing the authorization header is done through the usage of the “[authentication]” keyword. Depending on the algorithm (“MD5” or “AKAv1-MD5”), different parameters must be passed next to the authentication keyword:

- Digest/MD5 (example: `[authentication username=joe password=schmo]`)
 - username : username: if no username is specified, the username is taken from the ‘-au’ (authentication username) or ‘-s’ (service) command line parameter
 - password : password: if no password is specified, the password is taken from the ‘-ap’ (authentication password) command line parameter
- Digest/AKA: (example: `[authentication username=HappyFeet aka_OP=0xCDC202D5123E20F62B6D676AC72CB318 aka_K=0x465B5CE8B199B49FAA5F0A2EE238A6BC aka_AMF=0xB9B9]`)
 - username : username: if no username is specified, the username is taken from the ‘-au’ (authentication username) or ‘-s’ (service) command line parameter
 - aka_K : Permanent secret key. If no aka_K is provided, the “password” attributed is used as aka_K.
 - aka_OP : OPerator variant key
 - aka_AMF : Authentication Management Field (indicates the algorithm and key in use)

In case you want to use authentication with a different username/password or aka_K for each call, you can do this:

- Make a CSV like this:

```
SEQUENTIAL
User0001;[authentication username=joe password=schmo]
User0002;[authentication username=john password=smith]
User0003;[authentication username=betty password=boop]
```

- And an XML like this (the `[field1]` will be substituted with the full auth string, which is the processed as a new keyword):

```
<send retrans="500">
  <![CDATA[

    REGISTER sip:[remote_ip] SIP/2.0
    Via: SIP/2.0/[transport] [local_ip]:[local_port]
    To: <sip:[field0]@sip.com:[remote_port]>
    From: <sip:[field0]@[remote_ip]:[remote_port]>
    Contact: <sip:[field0]@[local_ip]:[local_port]>;transport=[transport]
    [field1]
```

(continues on next page)

(continued from previous page)

```

    Expires: 300
    Call-ID: [call_id]
    CSeq: 2 REGISTER
    Content-Length: 0

  ]]>
</send>

```

Example:

```

<recv response="407" auth="true">
</recv>

<send>
  <![CDATA[

    ACK sip:[service]@[remote_ip]:[remote_port] SIP/2.0
    Via: SIP/2.0/[transport] [local_ip]:[local_port]
    From: sipp <sip:sipp@[local_ip]:[local_port]>;tag=[call_number]
    To: sut <sip:[service]@[remote_ip]:[remote_port]>[peer_tag_param]
    Call-ID: [call_id]
    CSeq: 1 ACK
    Contact: sip:sipp@[local_ip]:[local_port]
    Max-Forwards: 70
    Subject: Performance Test
    Content-Length: 0

  ]]>
</send>

<send retrans="500">
  <![CDATA[

    INVITE sip:[service]@[remote_ip]:[remote_port] SIP/2.0
    Via: SIP/2.0/[transport] [local_ip]:[local_port]
    From: sipp <sip:sipp@[local_ip]:[local_port]>;tag=[call_number]
    To: sut <sip:[service]@[remote_ip]:[remote_port]>
    Call-ID: [call_id]
    CSeq: 2 INVITE
    Contact: sip:sipp@[local_ip]:[local_port]
    [authentication username=foouser]
    Max-Forwards: 70
    Subject: Performance Test
    Content-Type: application/sdp
    Content-Length: [len]

    v=0
    o=user1 53655765 2353687637 IN IP[local_ip_type] [local_ip]
    s=-
    t=0 0
    c=IN IP[media_ip_type] [media_ip]
    m=audio [media_port] RTP/AVP 0
    a=rtpmap:0 PCMU/8000

  ]]>
</send>

```

5.9 Initialization Stanza

Some complex scenarios require setting appropriate global variables at SIPp startup. The initialization stanza allows you do do just that. To create an initialization stanza, simply surround a series of `<nop>` and `<label>` commands with `<init>` and `</init>`. These `<nop>`s are executed once at SIPp startup. The variables within the init stanza, except for globals, are not shared with calls. For example, this init stanza sets `$THINKTIME` to 1 if it is not already set (e.g., by the `-set` command line parameter).

```
<init>
  <!-- By Default THINKTIME is true. -->
  <nop>
    <action>
      <strcmp assign_to="empty" variable="THINKTIME" value="" />
      <test assign_to="empty" compare="equal" variable="empty" value="0" />
    </action>
  </nop>
  <nop condexec="empty">
    <action>
      <assignstr assign_to="THINKTIME" value="1" />
    </action>
  </nop>
</init>
```


CHAPTER 6

3PCC Extended

An extension of the 3pcc mode is implemented in SIPp. This feature allows any number of SIPp instances to communicate with each other, each one of them being connected to a remote host.

The SIPp instance which initiates the call is launched in “master” mode. The others are launched in “slave” mode. Slave SIPp instances have names, given in the command line (for example, s1, s2...sN for the slaves and m for the master) Correspondances between instances names and their addresses must be stored in a file (provided by `-slave_cfg` command line argument), in the following format:

```
s1;127.0.0.1:8080
s2;127.0.0.1:7080
m;127.0.0.1:6080
```

Each SIPp instance must access a different copy of this file.

`sendCmd` and `recvCmd` have additional attributes:

```
<sendCmd dest="s1">
  <![CDATA[
    Call-ID: [call_id]
    From: m
    [$1]
  ]]>
</sendCmd>
```

Will send a command to the “s1” peer instance, which can be either master or slave, depending on the command line argument, which must be consistent with the scenario: a slave instance cannot have a `sendCmd` action before having any `recvCmd`. Note that the message must contain a “From” field, filled with the name of the sender.

```
<recvCmd src="m">
  <action>
    <ereg regexp="Content-Type:.*"
      search_in="msg"
      assign_to="2"/>
```

(continues on next page)

(continued from previous page)

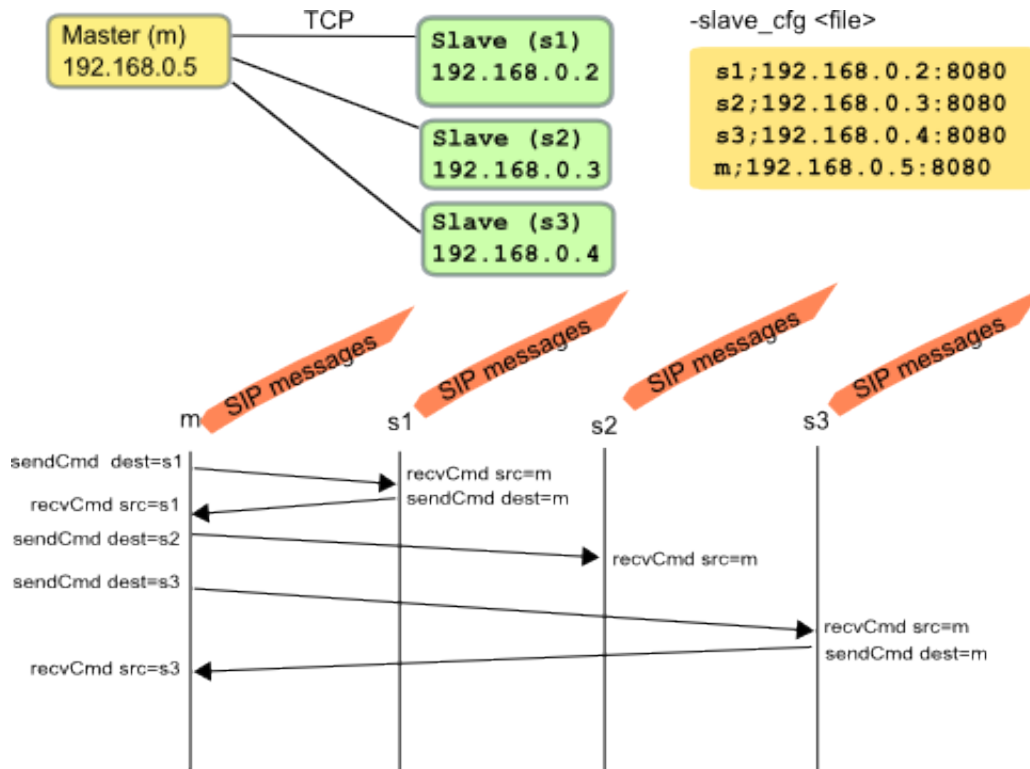
```
</action>
</recvCmd>
```

Indicates that the twin command is expected to be received from the “m” peer instance.

Note that the master must be the launched at last.

There is no integrated scenarios for the 3pcc extended mode, but you can easily adapt those from 3pcc.

Example: the following drawing illustrate the entire procedure. The arrows that are shown between SIPp master and slaves depict only the synchronization commands exchanged between the different SIPp instances. The SIP message exchange takes place as usual.



Controlling SIPp

SIPp can be controlled interactively through the keyboard or via a UDP socket. SIPp supports both ‘hot’ keys that can be entered at any time and also a simple command mode. The hot keys are:

Key	Action
+	Increase the call rate by $1 * \text{rate_scale}$
*	Increase the call rate by $10 * \text{rate_scale}$
-	Decrease the call rate by $1 * \text{rate_scale}$
/	Decrease the call rate by $10 * \text{rate_scale}$
c	Enter command mode
q	Quit SIPp (after all calls complete, enter a second time to quit immediately)
Q	Quit SIPp immediately s Dump screens to the log file (if -trace_screen is passed)
p	Pause traffic
1	Display the scenario screen
2	Display the statistics screen
3	Display the repartition screen
4	Display the variable screen
5	Display the TDM screen
6-9	Display the second through fifth repartition screen.

In command mode, you can type a single line command that instructs SIPp to take some action. Command mode is more versatile than the hot keys, but takes more time to input some common actions. The following commands are available:

7.1 List of Interactive Commands

- `dump tasks` Prints a list of active tasks (most tasks are calls) to the error log. `dump tasks`
- `set rate X` Sets the call rate. `set rate 10`
- `set rate-scale X` Sets the rate scale, which adjusts the speed of ‘+’, ‘-’, ‘*’, and ‘/’. `set rate-scale 10`

- `set users X` Sets the number of users (only valid when `-users` is specified). `set rate 10`
- `set limit X` Sets the open call limit (equivalent to `-l` option) `set limit 100`
- `set hide <true|false>` Should the hide XML attribute be respected? `set hide false`
- `set index <true|false>` Display message indexes in the scenario screen. `set index true`
- `set display <main|ooc>` Changes the scenario that is displayed to either the main or the out-of-call scenario. `set display main` `set display ooc`
- `trace <log> <on|off>` Turns log on or off at run time. Valid values for log are “error”, “logs”, “messages”, and “shortmessages”. `trace error on`

7.2 Traffic control

SIPp generates SIP traffic according to the scenario specified. You can control the number of calls (scenario) that are started per second. If you pass the `-users` option, then you need to control the number of instantiated users. You can control the rate through:

- Interactive hot keys (described in the previous section)
- Interactive Commands
- Startup Parameters

There are two commands that control rates: `set rate X` sets the current call rate to X. Additionally, `set rate-scale X` sets the `rate_scale` parameter to X. This enables you to use the ‘+’, ‘-’, ‘*’, and ‘/’ keys to set the rate more quickly. For example, if you do `set rate-scale 100`, then each time you press ‘+’, the call rate is increased by 100 calls and each time you press ‘*’, the call rate is increased by 1000 calls. Similarly, for a user based benchmark you can run `set users X`.

At starting time, you can control the rate by specifying parameters on the command line:

- “-r” to specify the call rate in number of calls per seconds
- “-rp” to specify the “rate period” in milliseconds for the call rate (default is 1000ms/1sec). This allows you to have n calls every m milliseconds (by using `-r n -rp m`).

Note: Example: run SIPp at 7 calls every 2 seconds (3.5 calls per second)

```
./sipp -sn uac -r 7 -rp 2000 127.0.0.1
```

You can also pause the traffic by pressing the ‘p’ key. SIPp will stop placing new calls and wait until all current calls go to their end. You can resume the traffic by pressing ‘p’ again.

To quit SIPp, press the ‘q’ key. SIPp will stop placing new calls and wait until all current calls go to their end. SIPp will then exit.

You can also force SIPp to quit immediately by pressing the ‘Q’ key. Current calls will be terminated by sending a BYE or CANCEL message (depending if the calls have been established or not). The same behaviour is obtained by pressing ‘q’ twice.

Tip: You can place a defined number of calls and have SIPp exit when this is done. Use the `-m` option on the command line.

7.3 Remote control

SIPp can be “remote-controlled” through a UDP socket. This allows for example

- To automate a series of actions, like increasing the call rate smoothly, wait for 10 seconds, increase more, wait for 1 minute and loop
- Have a feedback loop so that an application under test can remote control SIPp to lower the load, pause the traffic, ...

Each SIPp instance is listening to a UDP socket. It starts to listen to port 8888 and each following SIPp instance (up to 60) will listen to `base_port + 1` (8889, 8890, ...).

It is then possible to control SIPp like this:

```
echo p >/dev/udp/x.y.z.t/8888 -> put SIPp in pause state (p key)
echo q >/dev/udp/x.y.z.t/8888 -> quit SIPp (q key)
```

Note: All keys available through keyboard are also available in the remote control interface

You could also have a small shell script to automate a serie of action. For example, this script will increase the call rate by 10 more new calls/s every 5 seconds, wait at this call rate for one minute and exit SIPp:

```
#!/bin/sh
echo "*" >/dev/udp/127.0.0.1/8889
sleep 5
echo "*" >/dev/udp/127.0.0.1/8889
sleep 5
echo "*" >/dev/udp/127.0.0.1/8889
sleep 5
echo "*" >/dev/udp/127.0.0.1/8889
sleep 60
echo "q" >/dev/udp/127.0.0.1/8889
```

To send a command to SIPp, preface it with ‘c’. For example: `echo "cset rate 100" >/dev/udp/127.0.0.1/8888` sets the call rate to 100.

SIPp has several transport modes. The default transport mode is “UDP mono socket”.

8.1 UDP mono socket

In UDP mono socket mode (-t u1 command line parameter), one IP/UDP socket is opened between SIPp and the remote. All calls are placed using this socket.

This mode is generally used for emulating a relation between 2 SIP servers.

8.2 UDP multi socket

In UDP multi socket mode (-t un command line parameter), one IP/UDP socket is opened for each new call between SIPp and the remote.

This mode is generally used for emulating user agents calling a SIP server.

8.3 UDP with one socket per IP address

In UDP with one socket per IP address mode (-t ui command line parameter), one IP/UDP socket is opened for each IP address given in the inf file.

In addition to the “-t ui” command line parameter, one must indicate which field in the inf file is to be used as local IP address for this given call. Use “-ip_field <nb>” to provide the field number.

There are two distinct cases to use this feature:

- Client side: when using -t ui for a client, SIPp will originate each call with a different IP address, as provided in the inf file. In this case, when your IP addresses are in field X of the inject file, then you have to use [fieldX] instead of [local_ip] in your UAC XML scenario file.

- Server side: when using `-t ui` for a server, SIPp will bind itself to all the IP addresses listed in the `inf` file instead of using `0.0.0.0`. This will have the effect SIPp will answer the request on the same IP on which it received the request. In order to have proper Contact and Via fields, a keyword `[server_ip]` can be used and provides the IP address on which a request was received. So when using this, you have to replace the `[local_ip]` in your UAS XML scenario file by `[server_ip]`.

In the following diagram, the command line for a client scenario will look like: `./sipp -sf myscenario.xml -t ui -inf database.csv -ip_field 2 192.168.1.1` By doing so, each new call will come sequentially from IP 192.168.0.1, 192.168.0.2, 192.168.0.3, 192.168.0.1, ...

This mode is generally used for emulating user agents, using on IP address per user agent and calling a SIP server.

8.4 TCP mono socket

In TCP mono socket mode (`-t t1` command line parameter), one IP/TCP socket is opened between SIPp and the remote. All calls are placed using this socket.

This mode is generally used for emulating a relation between 2 SIP servers.

8.5 TCP multi socket

In TCP multi socket mode (`-t tn` command line parameter), one IP/TCP socket is opened for each new call between SIPp and the remote.

This mode is generally used for emulating user agents calling a SIP server.

8.6 TCP reconnections

SIPp handles TCP reconnections. In case the TCP socket is lost, SIPp will try to reconnect. The following parameters on the command line control this behaviour:

- `-max_reconnect` : Set the maximum number of reconnection attempts.
- `-reconnect_close true/false` : Should calls be closed on reconnect?
- `-reconnect_sleep int` : How long to sleep (in milliseconds) between the close and reconnect?

8.7 TLS mono socket

In TLS mono socket mode (`-t l1` command line parameter), one secured TLS (Transport Layer Security) socket is opened between SIPp and the remote. All calls are placed using this socket.

This mode is generally used for emulating a relation between 2 SIP servers.

Warning: When using TLS transport, SIPp will expect to have two files in the current directory: a certificate (`cacert.pem`) and a key (`cakey.pem`). If one is protected with a password, SIPp will ask for it.

SIPp supports X509's CRL (Certificate Revocation List). The CRL is read and used if `-tls_crl` command line specifies a CRL file to read.

8.8 TLS multi socket

In TLS multi socket mode (-t ln command line parameter), one secured TLS (Transport Layer Security) socket is opened for each new call between SIPp and the remote.

This mode is generally used for emulating user agents calling a SIP server.

8.9 SCTP mono socket

In SCTP mono socket mode (-t s1 command line parameter), one SCTP (Stream Transmission Control Protocol) socket is opened between SIPp and the remote. All calls are placed using this socket.

This mode is generally used for emulating a relation between 2 SIP servers.

The -multihome, -heartbeat, -assocmaxret, -pathmaxret, -pmtu and -gracefulclose command-line arguments allow control over specific features of the SCTP protocol, but are usually not necessary.

8.10 SCTP multi socket

In SCTP multi socket mode (-t sn command line parameter), one SCTP socket is opened for each new call between SIPp and the remote.

This mode is generally used for emulating user agents calling a SIP server.

8.11 IPv6 support

SIPp includes IPv6 support. To use IPv6, just specify the local IP address (-i command line parameter) to be an IPv6 IP address.

The following example launches a UAS server listening on port 5063 and a UAC client sending IPv6 traffic to that port.

```
./sipp -sn uas -i [fe80::204:75ff:fe4d:19d9] -p 5063
./sipp -sn uac -i [fe80::204:75ff:fe4d:19d9] [fe80::204:75ff:fe4d:19d9]:5063
```

Warning: The Pcap play feature may currently not work on IPv6.

8.12 Multi-socket limit

When using one of the “multi-socket” transports, the maximum number of sockets that can be opened (which corresponds to the number of simultaneous calls) will be determined by the system (see how to increase file descriptors section to modify those limits). You can also limit the number of socket used by using the -max_socket command line option. Once the maximum number of opened sockets is reached, the traffic will be distributed over the sockets already opened.

Handling media with SIPp

SIPp is originally a signalling plane traffic generator. There is a limited support of media plane (RTP).

9.1 RTP echo

The “RTP echo” feature allows SIPp to listen to one or two local IP address and port (specified using `-mi` and `-mp` command line parameters) for RTP media. Everything that is received on this address/port is echoed back to the sender.

RTP/UDP packets coming on this port + 2 are also echoed to their sender (used for sound and video echo).

9.2 RTP streaming

SIPp can play a PCMA, PCMU, G722, iLBC or G729-encoded audio file over RTP.

More details on how to do this can be found in the action reference section.

9.3 PCAP Play

The PCAP play feature makes use of the [PCAP library](#) to replay pre- recorded RTP streams towards a destination. RTP streams can be recorded by tools like Wireshark or `tcpdump`. This allows you to:

- Play any RTP stream (voice, video, voice+video, out of band DTMFs/[RFC 2833](#), T38 fax, ...)
- Use any codec as the codec is not handled by SIPp
- Emulate precisely the behavior of any SIP equipment as the pcap play will try to replay the RTP stream as it was recorded (limited to the performances of the system).
- Reproduce exactly what has been captured using an IP sniffer like Wireshark.

A good example is the UAC with media (uac_pcap) embedded scenario.

SIPp comes with a G711 alaw pre-recorded pcap file and out of band ([RFC 2833](#)) DTMFs in the pcap/ directory.

Warning: The PCAP play feature uses `pthread_setschedparam` calls from `pthread` library. Depending on the system settings, you might need to be root to allow this. Please check “`man 3 pthread_setschedparam`” man page for details

More details on the possible PCAP play actions can be found in the action reference section.

10.1 Response times

Response times can be gathered and reported. Response time names can be arbitrary strings, but for backwards compatibility the value “true” is treated as if it were named “1”. Each response time can be used to compute time between two SIPp commands (send, recv or nop). You can start a timer by using the start_rtd attribute and stop it using the rtd attribute.

You can view the value of those timers in the SIPp interface by pressing 3, 6, 7, 8 or 9. You can also save the values in a CSV file using the -trace_stat option (see below).

If the -trace_rtt option is set, the response times are also dumped in the >scenario file name<_pid<_rtt.csv.

Each line represents a RTD measure (triggered by a message reception with a rtd=”n” attribute). The dump frequency is tuned by the -rtt_freq parameter.

10.2 Available counters

The -trace_stat option dumps all statistics in the scenario_name_pid.csv file. The dump starts with one header line with all counters. All following lines are ‘snapshots’ of statistics counter given the statistics report frequency (-fd option). When SIPp exits, the last values of the statistics are also dumped in this file.

This file can be easily imported in any spreadsheet application, like Excel.

In counter names, (P) means ‘Periodic’ - since last statistic row and (C) means ‘Cumulated’ - since sipp was started.

Available statistics are:

- StartTime: Date and time when the test has started.
- LastResetTime: Date and time when periodic counters were last reseted.
- CurrentTime: Date and time of the statistic row.
- ElapsedTime: Elapsed time.

- **CallRate:** Call rate (calls per seconds).
- **IncomingCall:** Number of incoming calls.
- **OutgoingCall:** Number of outgoing calls.
- **TotalCallCreated:** Number of calls created.
- **CurrentCall:** Number of calls currently ongoing.
- **SuccessfulCall:** Number of successful calls.
- **FailedCall:** Number of failed calls (all reasons).
- **FailedCannotSendMessage:** Number of failed calls because Sipp cannot send the message (transport issue).
- **FailedMaxUDPRepeats:** Number of failed calls because the maximum number of UDP retransmission attempts has been reached.
- **FailedUnexpectedMessage:** Number of failed calls because the SIP message received is not expected in the scenario.
- **FailedCallRejected:** Number of failed calls because of Sipp internal error. (a scenario sync command is not recognized or a scenario action failed or a scenario variable assignment failed).
- **FailedCmdNotSent:** Number of failed calls because of inter-Sipp communication error (a scenario sync command failed to be sent).
- **FailedRegexDoesntMatch:** Number of failed calls because of regexp that doesn't match (there might be several regexp that don't match during the call but the counter is increased only by one).
- **FailedRegexShouldntMatch:** Number of failed calls because of regexp that shouldn't match (there might be several regexp that shouldn't match during the call but the counter is increased only by one).
- **FailedRegexHdrNotFound:** Number of failed calls because of regexp with `hdr` option but no matching header found.
- **FailedOutboundCongestion:** Number of failed outgoing calls because of TCP congestion.
- **FailedTimeoutOnRecv:** Number of failed calls because of a `recv` timeout statement.
- **FailedTimeoutOnSend:** Number of failed calls because of a `send` timeout statement.
- **OutOfCallMsgs:** Number of SIP messages received that cannot be associated with an existing call.
- **Retransmissions:** Number of SIP messages being retransmitted.
- **AutoAnswered:** Number of unexpected specific messages received for new Call-ID. The message has been automatically answered by a 200 OK. Currently, implemented for 'PING' message only.

The counters defined in the scenario are also dumped in the stat file. Counters that have a numeric name are identified by the `GenericCounter` columns.

In addition, two other statistics are gathered:

- **ResponseTime** (see previous section)
- **CallLength:** this is the time of the duration of an entire call.

Both `ResponseTime` and `CallLength` statistics can be tuned using `ResponseTimeRepartition` and `CallLengthRepartition` commands in the scenario.

The standard deviation (`STDev`) is also available in the log stat file for these two statistics.

10.3 Detailed Message Counts

The SIPp screens provide detailed information about the number of messages sent or recieved, retransmissions, messages lost, and the number of unexpected messages for each scenario element. Although these screens can be parsed, it is much simpler to parse a CSV file. To produce a CSV file that contains the per-message information contained in the main display screen pass the `-trace_counts` option. Each column of the file represents a message and a particular count of interest (e.g., “1_INVITE_Sent” or “2_100_Unexp”). Each row corresponds to those statistics at a given statistics reporting interval.

SIPp has advanced feature to handle errors and unexpected events. They are detailed in the following sections.

11.1 Unexpected messages

- When a SIP message that can be correlated to an existing call (with the Call-ID: header) but is not expected in the scenario is received, SIPp will send a CANCEL message if no 200 OK message has been received or a BYE message if a 200 OK message has been received. The call will be marked as failed. If the unexpected message is a 4XX or 5XX, SIPp will send an ACK to this message, close the call and mark the call as failed.
- When a SIP message that can't be correlated to an existing call (with the Call-ID: header) is received, SIPp will send a BYE message. The call will not be counted at all.
- When a SIP "PING" message is received, SIPp will send an ACK message in response. This message is not counted as being an unexpected message. But it is counted in the "AutoAnswered" statistic counter.
- An unexpected message that is not a SIP message will be simply dropped.

11.2 Retransmissions (UDP only)

A retransmission mechanism exists in UDP transport mode. To activate the retransmission mechanism, the "send" command must include the "retrans" attribute.

When it is activated and a SIP message is sent and no ACK or response is received in answer to this message, the message is re-sent.

Note: The retransmission mechanism follows [RFC 3261](#), section 17.1.1.2. Retransmissions are differentiated between INVITE and non-INVITE methods.

<send retrans="500">: will initiate the T1 timer to 500 milliseconds.

Even if retrans is specified in your scenarios, you can override this by using the `-nr` command line option to globally disable the retransmission mechanism.

11.3 Log files

There are several ways to trace what is going on during your SIPp runs.

- You can log sent and received SIP messages in `<name_of_the_scenario>_<pid>_messages.log` by using the command line parameter `-trace_msg`. The messages are time-stamped so that you can track them back.
- You also can trace it using the `-trace_shortmsg` parameter. This logs the most important values of a message as CSV into one line of the `<scenario file name>_<pid>_shortmessages.log`
- You can trace all unexpected messages or events in `<name_of_the_scenario>_<pid>_errors.log` by using the command line parameter `-trace_err`.
- You can trace the SIP response codes of unexpected messages in `<name_of_the_scenario>_<pid>_error_codes.log` by using the command line parameter `-trace_error_codes`.
- You can trace the counts from the main scenario screen in `<name_of_the_scenario>_<pid>_counts.csv` by using the command line parameter `-trace_counts`.
- You can trace the messages and state transitions of failed calls in `<name_of_the_scenario>_<pid>_calldebug.log` using the `-trace_calldebug` command line parameter. This is useful, because it has less overhead than `-trace_msg` yet allows you to debug call flows that were not completed successfully.
- You can save in a file the statistics screens, as displayed in the interface. This is especially useful when running SIPp in background mode. This can be done in different ways:
 - When SIPp exits to get a final status report (`-trace_screen` option)
 - On demand by using `USR2` signal (example: `kill -SIGUSR2 738`)
 - By pressing ‘s’ key (if `-trace_screen` option is set)
 - If the `-trace_logs` option is set, you can use the `<log>` action to print some scenario traces in the `<scenario file name>_<pid>_logs.log` file. See the Log action section

SIPp can treat the messages, short messages, logs, and error logs as ring buffers. This allows you to limit the total amount of space used by these log files and keep only the most recent messages. To set the maximum file size use the `-ringbuffer_size` option. Once the file exceeds this size (the file size can be exceeded up to the size of a single log message), it is rotated. SIPp can keep several of the most recent files, to specify the number of files to keep use the `-ringbuffer_files` option. The rotated files have a name of the form `<name_of_the_scenario>_<pid>_<logname>_<date>.log`, where `<date>` is the number of seconds since the epoch. If more than one log file is rotated during a one second period, then the date is expressed as `<seconds.serial>`, where `serial` is an increasing integer identifier.

12.1 Advice to run performance tests with SIPp

SIPp has been originally designed for SIP performance testing. Reaching high call rates and/or high number of simultaneous SIP calls is possible with SIPp, provided that you follow some guidelines:

- Use a Linux system to reach high performances. The Windows port of SIPp (through CYGWIN) cannot handle high performances.
- Limit the traces to a minimum (usage of `-trace_msg`, `-trace_logs` should be limited to scenario debugging only)
- Understand internal SIPp's scheduling mechanism and use the `-timer_resol`, `-max_recv_loops` and `-max_sched_loops` command line parameters to tune SIPp given the system it is running on.

Generally, running performance tests also implies measuring response times. You can use SIPp's timers (`start_rtd`, `rtd` in scenarios and `-trace_rtt` command line option) to measure those response times. The precision of those measures are entirely dependent on the `timer_resol` parameter (as described in *SIPp's internal scheduling* section). You might want to use another “objective” method if you want to measure those response times with a high precision (a tool like Wireshark will allow you to do so).

12.2 SIPp's internal scheduling

SIPp has a single-threaded event-loop architecture, which allows it to handle high SIP traffic loads. SIPp's event loop tracks various tasks, most of which are the calls that are defined in your scenario. In addition to tasks that represent calls there are several special tasks: a screen update task, a statistics update task, a call opening task, and a watchdog task. SIPp's main execution loop consists of:

1. Waking up tasks that have expired timers.
2. Running up to `max_sched_loop` tasks that are in a running state (each call is executed until it is no longer runnable).
3. Handling each of the sockets in turn, reading `max_recv_loops` messages from the various sockets.

SIPp executes this loop continuously, until some condition tells it to stop (e.g., the user pressing the ‘q’ key or the global call limit or timeout being reached).

Several parameters can be specified on the command line to fine tune this scheduling.

- `timer_resol`: during the main loop, the management of calls (management of wait, retransmission ...) is done for all calls, every “`timer_resol`” ms at best. The delay of retransmission must be higher than “`timer_resol`”. The default timer resolution is 1 millisecond, and that is the most precise resolution that SIPp currently supports. If you increase this parameter, SIPp’s traffic will be burstier and you are likely to encounter retransmissions at high load. If you have too many calls, or each call takes too long, the timer resolution will not be respected.
- `max_rcv_loops` and `max_sched_loops`: received messages are read and treated in batch. “`max_rcv_loops`” is the maximum number of messages that can be read at one time. “`max sched loops`” is the maximum number of processing calls loops. These limits prevent SIPp from reading and processing new messages from sockets to the exclusion of processing existing calls, and vice versa. For heavy call rate, increase both values. Be careful, those two parameters have a large influence on the CPU occupation of SIPp.
- `watchdog_interval`, `watchdog_minor_threshold`, `watchdog_major_threshold`, `watchdog_minor_maxtriggers`, and `watchdog_major_maxtriggers`: The watchdog timer is designed to provide feedback if your call load is causing SIPp’s scheduler to be overwhelmed. The watchdog task sets a timer that should fire every `watchdog_interval` milliseconds (by default 400ms). If the timer is not serviced for more than `watchdog_minor_threshold` milliseconds (by default 500s), then a “minor” trigger is recorded. If the number of minor triggers is more than `watchdog_minor_maxtriggers`; the watchdog task terminates SIPp. Similarly, if the timer is not serviced for more than `watchdog_major_threshold` milliseconds (by default 3000ms), then a major trigger is recorded; and if more than `watchdog_major_maxtriggers` are recorded SIPp is terminated. If you only see occasional messages, your test is likely acceptable, but if these events are frequent you need to consider using a more powerful machine or set of machines to run your scenario.

13.1 JEdit

JEdit is a GNU GPL text editor written in Java, and available on almost all platforms. It's extremely powerful and can be used to edit SIPp scenarios with syntax checking if you put the DTD ([sipp.dtd](#)) in the same directory as your XML scenario.

13.2 Wireshark/tshark

Wireshark is a GNU GPL protocol analyzer. It was formerly known as Ethereal. It supports SIP/SDP/RTP.

13.3 SIP callflow

When tracing SIP calls, it is very useful to be able to get a call flow from an wireshark trace. The “callflow” tool allows you to do that in a graphical way: [callflow](#)

An equivalent exist if you want to generate HTML only call flows <http://www.iptel.org/~sipsc/>

CHAPTER 14

Indices and tables

- `genindex`
- `modindex`
- `search`

R

RFC

RFC 2833, [59](#), [60](#)

RFC 3261, [20](#), [65](#)

RFC 3725, [14](#)