SIPp Extension

Table of Contents

Introduction 3

Core classes 4

Class hierarchy 4

SIPP(sipp.cpp) 4

SIPpSocket 5

Scenario 5

Call 5

**Listener** 5

**Socketowner** 5

Task 5

**Wheel** 5

SIP\_Parser 6

XP\_Parser 6

Core Tasks 6

Timewheel 6

Key API of time wheel 6

Ratetask 7

Public functions: 7

XML scenarios 9

Actions 10

Regular expressions 11

Log a message 12

Execute a command 13

Internal commands 13

External commands 13

Media/RTP commands 13

Variable Manipulation 14

String Variables 15

Variable Testing 15

lookup 15

Updating In-Memory Injection files 15

Jumping to an Index 16

setdest 16

verifyauth 17

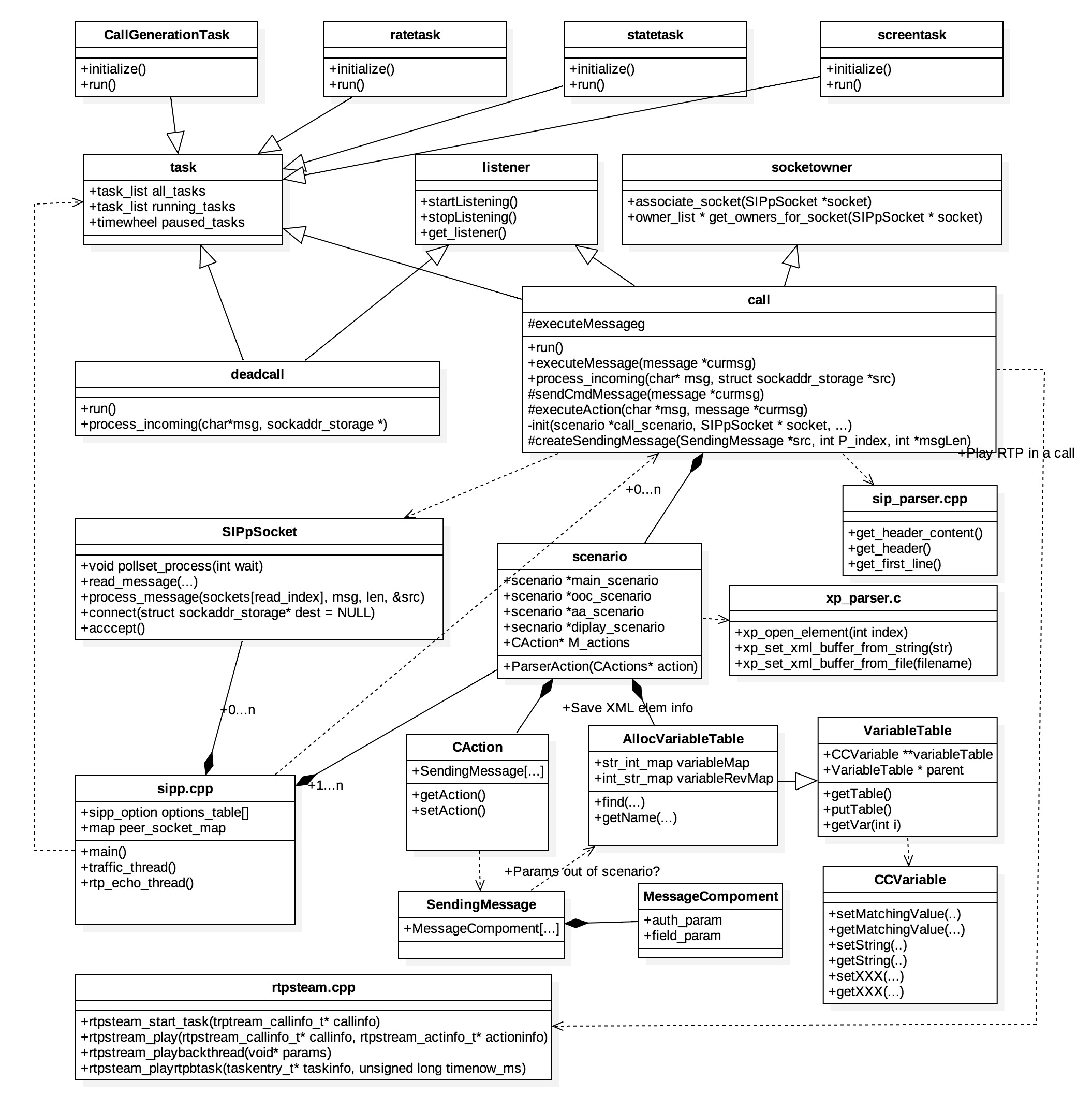
# Introduction

SIPp is a tool for testing SIP protocol performance, which contains some basic SipStone user agent workflows (UAC and UAS), and can use INVITE and BYE to create and release multiple calls. It can also read XML scene files, that is, describe any performance test configuration file (which can be used to simulate the scene of the SIP signaling to reproduce the fault; or you can customize the SIP protocol to test the terminal for some aspects of the Fault tolerance or error handling). It can dynamically display the test run statistics (call rate, signal back and forth delay, and message statistics). Periodically dumping CSV statistics, TCP and UDP over multiple sockets, and multiplexing using retransmission management. You can use the regular expression in the scene definition file to dynamically adjust the call rate. SIPp can be used to test many real SIP devices such as SIP proxy, B2BUAs, SIP media servers, SIP / x gateways, SIP PBXs, and so on, and it can also mimic thousands of SIP agents to call your SIP system. In addition, SIPp can also be used to test the SIP protocol.

# Core classes

## Class hierarchy

This chapter will show the class diagram of SIPp from top level. Hopefully this can help you reading the code more easily.



Below will brief introduce some main file or class according to above diagram.

## **SIPP(sipp.cpp)**

This is the file where main function located. The main function is responsible for all the initializations before sipp process start, parsing the command line ,setup socket connection ,launch rtp echo thread and process kinds of tasks in loop. So from above diagram, it composed by or depend on below class objects:

* SIPpSocket
* Scenario
* Call
* Task

## **SIPpSocket**

This is call provide the socket create, connect, read, write and so on operations relate to socket.

## **Scenario**

This is a class to store the kinds of parameters parsed from XML file or embedded scenarios. The scenario types may four types like below:

scenario \*main\_scenario; // created with the default

scenario \*ooc\_scenario; // out of a call senario

scenario \*aa\_scenario; // auto answer scenario

scenario \*display\_scenario; // Just for statistics of calls

## **Call**

This is a class inherited from task/listener/socketowner. It designed to handle the command initiated locally and process the message from network.

### **Listener**

This is a class used to help to match Call object with call id in the sip message from network. If the call object is matched, then invoke the method to process network meesage.

### **Socketowner**

This is a class to manager all the socket relates to one specific call object. To be more specific, it will map all the socket used in a call ot the call object so that can do socket close/clean relate things after a call is ended.

## **Task**

This class will maintain a list for all the tasks including callgenerationtask,call,ratetask,statetask and screetask. Besides, it declared a pure virtual function run() which will be implemented by each child class. The run() function will do what the task really need to do. Also, it will leverage wheel class to extend timer function.

### **Wheel**

The actual wheels. This is a variation on having one wheel for seconds, another for minutes and a third for hours - in this model, the first wheel holds tasks that should be scheduled in the next 2^12ms (~4s), the second wheel holds tasks that should be scheduled between 2^12 and 2^22ms (~4s-~69m), and the third wheel holds tasks that should be scheduled between 2^22ms and 2^32ms (~69m-~8 years).

## **SIP\_Parser**

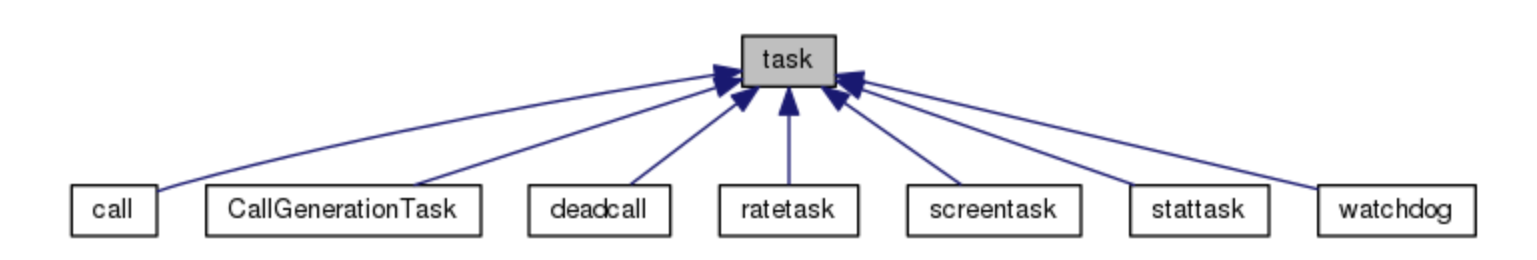
This is used to parse sip message. All the elements in sip message will be parsed and stored in this class.

## **XP\_Parser**

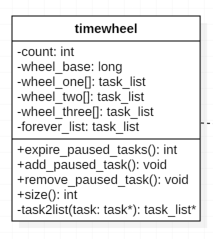
This is a class encapsulate all the methods to parse xml file in which defined call scenarios. The elements parsed from it will be stored in variableTable.

# Core Tasks

Task is a basic class in SIPP. Call, CallGenerationTask, deadcall, ratetask, screentask,stattask and watchdog.



The core idea of task class is a timer, in task class the timer is implemented in his friend class timewheel, which keep all the information about time.



## Timewheel

Time wheel takes O(1) time to start, stop, and maintain timers within the range of the wheel. Three extensions for larger values of the interval are described. In the first, the timer interval is hashed into a slot on the timing wheel. In the second and third, a hierarchy of timing wheels with different granularities is used to span a greater range of intervals. The performance of these three schemes and various implementation tradeoffs are discussed.

### Key API of time wheel

void task::recalculate\_wheel()

This function calculate the time of paused tasks and put the tasks in to the correct time in timewheel

task\_list \*timewheel::task2list(task \*task)

Based on the time a given task should next be woken up, finds the correct time wheel for it and returns a list of other tasks occurring at that point.

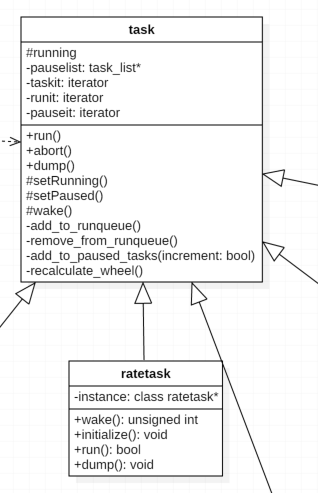
int timewheel::expire\_paused\_tasks()

Iterate through our sorted set of paused tasks, removing those that should no longer be paused, and adding them to the run queue.

void timewheel::add\_paused\_task(task \*task, bool increment)

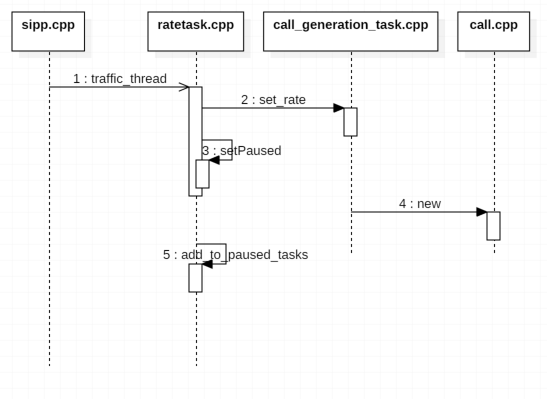
Adds a task to the correct timewheel. When increment is false, does not increment the count of tasks owned by this timewheel, and so can be used for recalculating the wheel of an existing task.

## Ratetask



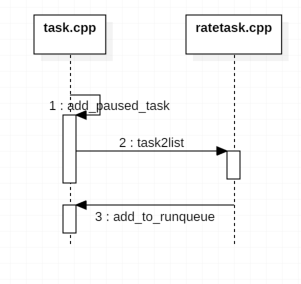
### Public functions:

Run()



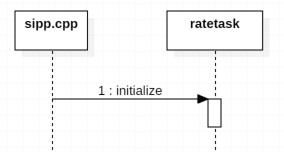
The traffic\_thread schedule all the task in running status

Wake()



Wake() function will put the task to runqueue which will be in running status

Initialize()



Sipp initialize the rate task in main function. Then the ratetask will be wake or run by the command

# XML scenarios

The xml scenario parser is realized in the constructor of the class scenario. The function tries to get xml script from input file, if there is not input file, it will use the xml default.

Take the “uac.xml” from <https://sipp.sourceforge.io/doc/uac.xml.html> as example.

SIPp UAC Remote

|(1) INVITE |

|------------------>|

|(2) 100 (optional) |

|<------------------|

|(3) 180 (optional) |

|<------------------|

|(4) 200 |

|<------------------|

|(5) ACK |

|------------------>|

| |

|(6) PAUSE |

| |

|(7) BYE |

|------------------>|

|(8) 200 |

|<------------------|

1. Check if the xml file has the header “ <?xml version="1.0" encoding="ISO-8859-1" ?>”;
2. skip the doctype line “<!DOCTYPE scenario SYSTEM "sipp.dtd">”;
3. skip the commention line “<!-- …… -->”;
4. check if the xml file has at least one element “<………..>”;
5. dump the scenario name from the line “<scenario name="Basic Sipstone UAC">” by searching “scenario name”;
6. get and parse the content of the invite send message which starts with “<send retrans="500">” and end with “</send>”
7. get the header “<send retrans="500">” of the send message by searching the key word “send”. retrans="500" will initiate T1 timer to 500 milliseconds for udp.
8. set the message type as “MSG\_TYPE\_SEND” for the function.
9. get the content of the message by searching starting key word "<![CDATA[" and ending key word “]]>”
10. call generation task will generate invite or ack message according to the key word “INVITE” and “ACK”. Here is invite message
11. call generation task will build message packet according to the key word in the type of “[…..]” such as “[remote\_ip]” , “[remote\_port]”, “[local\_ip]”, “[local\_port]” and etc.
12. get the value of the sending attribute “retrans” from “<send retrans="500">” which means that it will initiate T1 timer to 500 milliseconds for udp. Sending message also has the attribute “lost”, “start\_txn” and “ack\_txn” except “retrans”.
13. get and parse the content of the send message which starts with “<recv response="100" optional="true">” and end with “</recv>”.
14. “<recv response="100" optional="true">” means that the 100 SIP message can be received without being considered as "unexpected".
15. 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.
16. get and parse the content of the send message which starts with “<recv response="180" optional="true">” and end with “</recv>”. This means that the 180 SIP message can be received without being considered as "unexpected".
17. get and parse the content of the send message which starts with “<recv response="200" rtd="true">” and end with “</recv>”. This means that the 200 SIP message must be received, and the T1 timer will stop when the message is received.
18. get and parse the content of the ack message which starts with “<send>” and end with “</send>”
19. get the header of the send message by searching the key word “send”.
20. set the message type as “MSG\_TYPE\_SEND” for the function.
21. get the content of the message by searching starting key word "<![CDATA[" and ending key word “]]>”
22. call generation task will generate invite or ack message according to the key word “INVITE” and “ACK”. Here is ack message
23. call generation task will build message packet according to the key word in the type of “[…..]” such as “[remote\_ip]” , “[remote\_port]”, “[local\_ip]”, “[local\_port]” and etc.
24. “<pause/>” will insert a delay. “timewait” also has the same function.
25. get and parse the content of the bye message which starts with “<send retrans="500">” and end with “</send>”, the bye message parsing process is the same as the bye message.
26. get and parse the content of the send message which starts with “<recv response="200" crlf="true">” and end with “</recv>”. This means that the 200 SIP message must be received, and crlf="true">” means that it will display an empty line after the arrow for the message in main SIPp screen.
27. get and parse the content of the attribute “<ResponseTimeRepartition value="10, 20, 30, 40, 50, 100, 150, 200"/>” . This means that response times values are distributed between 0 and 10ms, 10 and 20ms, 20 and 30ms, 30 and 40ms, 40 and 50ms, 50 and 100ms, 100 and 150ms, 150 and 200ms, 200 and beyond.
28. get and parse the content of the attribute “<CallLengthRepartition value="10, 50, 100, 500, 1000, 5000, 10000"/>”. This means that call length time values are distributed between 0 and 10ms, 10 and 50ms, 50 and 100ms, 100 and 500ms, 500 and 1000ms, 1000 and 5000ms, 5000 and 10000ms, 10000 and beyond.
29. “</scenario>” means the end of the xml scenario file.

# 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

## **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 ` regexp tutorial`\_.

Here is the syntax of the regexp action:

**regexp action syntax**

|  |  |  |
| --- | --- | --- |
| **Keyword** | **Default** | **Description** |
| regexp | None | Contains the regexp 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\_indep | 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 regexp 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 regexp 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 regexp 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 Contract: 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>

## **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.

Notes:

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]"/>

## **Execute a command**

The “exec” action allows you to execute “internal”, “external”, “play\_pcap\_audio” or “play\_pcap\_video” commands.

## **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>

## **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>

## **Media/RTP commands**

RTP streaming allows you to stream audio from a PCMA, PCMU 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, and 18 indicates G729).
* <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](http://www.tcpdump.org/pcap3_man.html). 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

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.

## **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>

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>

## **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 lexographically 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>

## **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>

## **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>

</nop>

## **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>

## **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" />

  </action>

</nop>

**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>

## **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 baed 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.

## **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:]**

**[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="47ebe028cda119c35d4877b383027d28da013815"**

**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" />