

SIP Inspector - version 7



About:

In the beginning I needed a tool like SIP Inspector. After using various open source programs I started working on a better and simpler to use program. I hoped I succeeded. Version 6 added secure RTP, G.729 and multiple dialogs in one scenario. For version 7 I wanted to go further: the program now sends and decodes video, it can turn a Wireshark capture into a scenario, and it has a new look with a dark and a light theme. The command line mode is a first-class citizen too, so everything in this manual is shown both ways, in the GUI and on the command line.

Thanks for your encouragement and suggestions!

Žarko

Version 7.00 - October 2026

Table of Contents

Release notes 7.0	4
Commonly used terms and basic ideas	5
Starting the tool	6
Buying the license	7
Command Line Interface	8
The REST API step by step	15
Initial screen	25
Scenario building screen	26
SIP tab	28
RTP	30
Condition and branching	32
Command	34
Variables	35
Video	37
TLS	39
Scenario progress screen	41
Menu options	42
Scenario files, loading and saving	45
Run reports	48
Values files: one scenario, many users	55
SIP message handling	59
Multiple dialogs in a scenario	59
Option 'Don't check Call-ID'	61
Unconditional RTP streaming	61
Multipart SDP bodies	61
Hexadecimal bodies	62
3rd Party Call Control	62
Call generation	63
IVR testing	64
Trace to Scenario	64
Writing scenarios with Claude	69
Public/private keys	69
Self-signed certificate	70
SDP calculator	70
Logging and troubleshooting	71
Sample scenarios	72
Testing the program	74
Version checker	75

Keywords	75
Performance and capacity	76

Release notes 7.0

New features:

- **Video.** SIP Inspector can offer, send and receive H.264 video. Video is sent from a pcap file or from an MP4 file. Incoming video can be decoded and shown in the new **Video** tab, in a window of its own, and from the scenario screen. A scenario can wait until a number of pictures has been decoded (**Wait video**). New keyword: [video_port]. New command line options: -pv and -nv.
- **Trace to Scenario.** A Wireshark capture (pcap or pcapng) becomes a scenario: pick the endpoint that SIP Inspector takes the place of and the dialogs to include, and the program writes the scenario and the RTP that the endpoint sent. Available in the GUI (Tools/Trace to Scenario) and on the command line (-pcap2scen). Each dialog can keep its Call-ID as captured.
- **New commands in the Command tab:** Wait RTP, Wait video and Set variable.
- **Values files.** Use Each Values Row Once (Options menu, -fvo) makes exactly one call per line of the values file. Values can count down ({100-5}) and keep leading zeros, a ; can be part of a value (\;), every call keeps its own values, and an edited values file is read again on Start.
- **Validate Scenario.** Tools/Validate Scenario checks the SIP messages of a scenario for common mistakes (missing headers, tags, CSeq, Call-ID, Max-Forwards and the SDP) and marks the entries and the header lines that need attention.
- **Run reports and CI.** HTML, CSV, JUnit XML and JSON reports with the ASR, ACD, response times (p50, p95, p99) and RTP quality (loss, jitter, MOS); assertions in a scenario (assert=status, responseTime, header, body, rtp); a CI gate (-gate) whose limits decide the result and the exit code. Samples are in the samples/reports folder.
- **Control over REST.** -api opens an HTTP API to watch a run, read its numbers (also for Prometheus), change its rate and stop it. Without -fs the program is a service that takes run after run (POST /run) and keeps their reports. A control page for a web browser (/ui) shows the run live and has buttons for the rate, Stop, the reports and new runs. samples/api/si_suite.py runs a regression suite through the API.
- **Call flow.** The messages of a call as a ladder diagram, live in the scenario screen (Call Flow) and in the HTML report, with the first call that did not reach the end of the scenario kept for the whole run.
- **SIP and SDP checks.** Every message sent and received is checked against RFC 3261, RFC 4566 and the offer/answer rules of RFC 3264. What is found is a warning in the reports and the Checks button; a gate problems==0 fails on it. -noChecks turns the checks off.
- **Diagnostics bundle.** Help/Save Diagnostics, -diag and GET /diag: one zip with the settings, the scenario, the logs, the reports and the call flows, passwords masked, to send with a support question.
- **New look.** Dark and light themes (Options/Theme), rounded buttons, a banner, light yellow tool tips and windows that adapt when the theme is changed. The scenario entries screen can be made wider with a divider between the message tree and the tabs.
- **Scenario comments are kept.** Lines starting with // in a scenario file survive editing and saving. All scenarios that come with the program are commented and follow the SIP RFCs; new samples cover call transfer, video, SUBSCRIBE and BLF, PRACK, mid-call codec change, WebSocket (WebRTC) and more.
- **Logging reorganized.** One program log, SIPInspector.log, with four levels (ERROR, WARN, DEBUG, VERBOSE).
- **Licensing.** A new serial number scheme for version 7. The Machine-ID stays the same when network adapters are added or removed. The text of licensing errors can be selected and copied.
- **Security tools.** Key length is validated (RSA, DSA, DH and EC), a certificate that does not match its private key is refused and a damaged truststore is never overwritten.
- **Performance.** RTP from pcap files is sent on time and spread over several CPU cores (32 and more streams), the progress table is refreshed on a 250 ms timer so a fast run does not flood the user interface, and the check for a new version runs in the background so the main window shows at once.
- **Options/Don't check Call-ID** remembers your Calls/sec and Max Concurrent values and puts them back when the option is turned off.

- **Command line:** exit codes that scripts can use (0, 1 or 2), -logScen, -pv, -nv, -pcap2scen, and scenario pair tests (test/scenario_test.py).

Requirements: Java 17 or higher is needed (version 6 needed Java 8).

Bug fixes: a session ID that exceeded the maximum long number, extra characters when a new outgoing message was added, an error when a video call was made, the media port not being opened when a scenario waits for RTP, outgoing SRTP and RTP to several clients, and many more.

Commonly used terms and basic ideas

SIP Inspector is intended to troubleshoot and test SIP signalling. This requires the user to create a 'scenario'. A scenario can be seen as a set of 'scenario entries'. A scenario entry can be a SIP message, a label, a command, a pause, a variable or another available scenario entry. Entries are placed in the wanted order and, as scenario execution progresses, are executed one after another. In some cases an incoming message may or may not come. Such messages are **optional** and are marked with an asterisk. Conditional branching is possible with a 'goto', which transfers execution to a point in the scenario that is marked with a given label.

Just like when writing source code, try never to use hard-coding. Instead of writing From and To values like '1000' or '2000', use [fromUsername] and [toUsername]. These are called **keywords**. Before the program sends a message out, all keywords are replaced with their values. Most of the time one scenario represents signalling between an A and a B party. A single scenario can also be repeated for A and B, C and D, E and F, ... For that a **values file** is used: it holds values that are injected into the scenario, so the same scenario runs with different values.

Initially SIP Inspector was conceived to simulate only signalling. RTP streaming was added later: first RTP played from a pre-captured pcap file, then live speech (G.711 and G.729), then secure RTP (SRTP) and, in version 7, video.

SIP Inspector offers the following scenario execution possibilities:

- the scenario is executed once. To execute it again a user action is required;
- when finished, the scenario is automatically repeated a given number of times;
- the scenario is executed a given number of times, at a given call speed and in parallel;
- every time the scenario is executed the values of certain fields change. For example, the From and To headers differ for every new call that is generated.

The program runs in either **GUI** or **CLI** mode. Scenario files are plain text and are the same in both modes, so a scenario that was built and tried in the GUI can be run unattended from a script. The progress of a run, the SIP messages and the program log can be saved to files, which helps to debug problems.

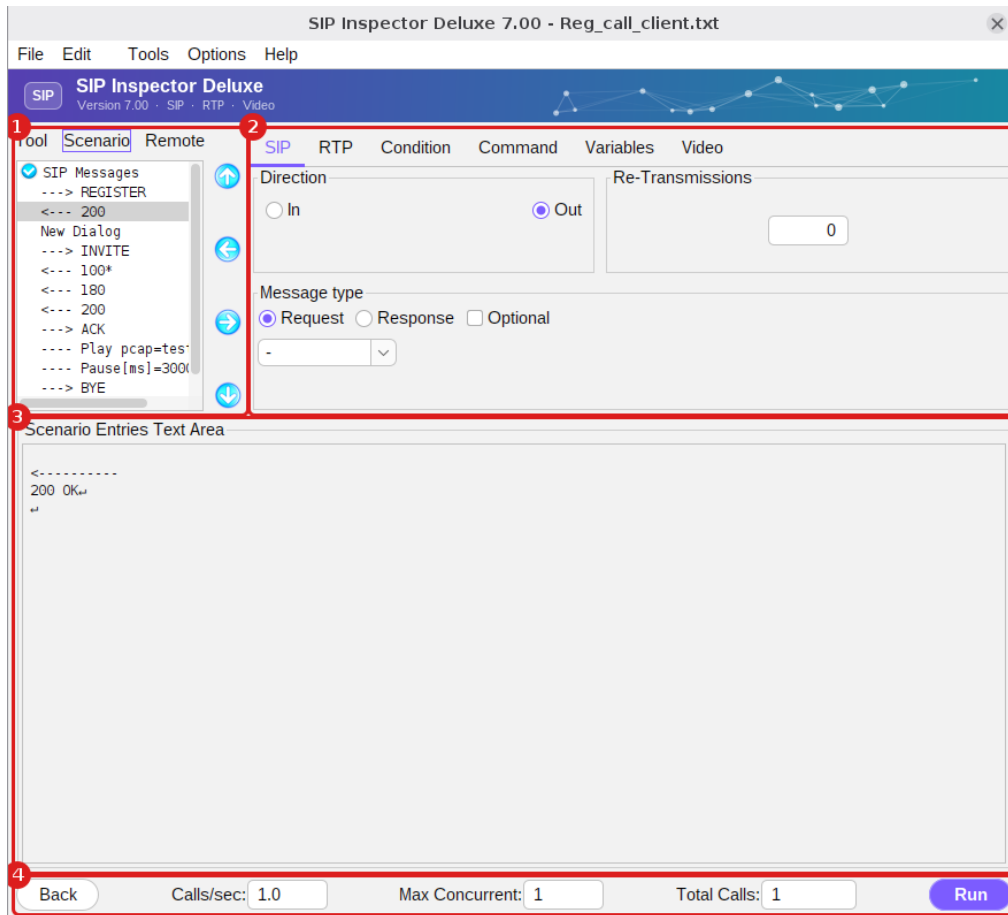


Figure 1 - The scenario screen of the GUI: (1) scenario entries, (2) scenario creation tools, (3) scenario entry text, (4) call generation and Run

Starting the tool

To run version 7 of SIP Inspector a JRE (Java Runtime Environment) 17 or higher must be installed. The program is offered for Windows and for other operating systems (Linux and macOS). Unpack the zip file and keep the folders together: scenarios, pcap and values hold the samples, and the program creates security (keys and certificates) and Logs next to the executable when it needs them.

Windows:

- SIPInspDlx.exe - double click it, or execute it, to get the GUI.
- SIPInspDlxCLI.exe - the command line mode. Run it from a command prompt, with options.

Other operating systems: ./SIPInspDlx starts the GUI without options and the CLI when options are given.

SIP Inspector Deluxe 7.00

File Edit Tools Options Help

SIP SIP Inspector Deluxe
Version 7.00 · SIP · RTP · Video

Account

From: 1000 Auth username: 1000

To: 2000 Auth password: secret1000

RTP Port: 6000

License Info

Machine ID: 7Gx2KqPz9Lw3TnVb5RhY8d

Serial Number: Kc4mQ9wXz2LpR7vTn1sBe6

Local Info

Select local IP address: 192.168.2.10

Local port: 5060

IP Version: ☒ IPv4 ☐ IPv6

Protocol: ☒ UDP ☐ TCP ☐ WS ☐ WSS ☐ TLS

Remote Server

☒ Remote Server

Remote Server: 192.168.1.98

Remote Server Port: 5060

Remote Path: /ws

Next

Figure 2 - The initial screen, after the program starts in GUI mode

On Windows 64-bit a message like the following may appear. In that case run the program as administrator.

WARNING: Could not open/create prefs root node Software\JavaSoft\Prefs at root 0x80000002. Windows RegCreateKeyEx(...) returned error code 5.

Buying the license

SIP Inspector Pro is paid-for software. To use it a license must be bought to acquire a serial number that unlocks the program. Serial numbers can be purchased through the web site. When buying a license please make sure to include your **Machine-ID**. The Machine-ID is reported by the program. Please copy and paste the value rather than typing it in. Version 7 uses a new serial number scheme: request a serial number for version 7 with your Machine-ID.

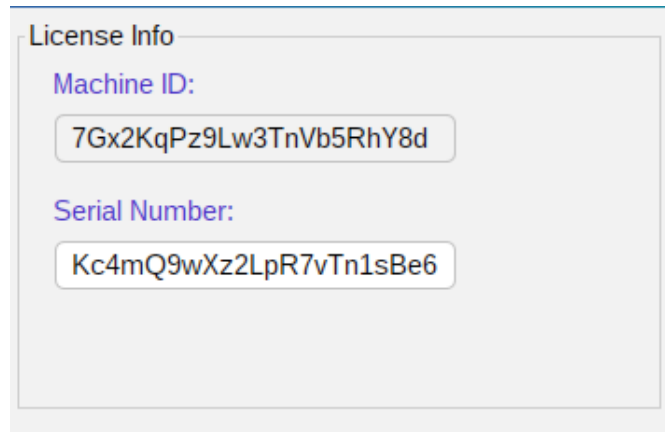


Figure 3 - Machine-ID and serial number (the values shown here are made up)

The Machine-ID is calculated from the computer: its network adapters and its operating system installation. Version 7 keeps it stable when network adapters are added or removed, so a license does not stop working because of a new Wi-Fi or USB network adapter. A new installation of the operating system, or another computer or virtual machine, has another Machine-ID and needs a serial number of its own. A serial number of an earlier release that does not yet include the operating system installation is refused; the program then shows the Machine-ID to send for a new one.

Use the command line to get the Machine-ID and to enter the serial number:

```

Terminal
$ SIPInspDlx -m
Machine-ID: 7Gx2KqPz9Lw3TnVb5RhY8d
$ SIPInspDlx -sn Kc4mQ9wXz2LpR7vTn1sBe6
SUCCESS
$ SIPInspDlx -v
Version: SIP Inspector Deluxe 7.00

```

Figure 4 - Machine-ID, serial number and version on the command line (Linux or macOS; on Windows use SIPInspDlxCLI.exe)

Windows examples:

```

C:\SI\windows>SIPInspDlxCLI.exe -m
Machine-ID: 7Gx2KqPz9Lw3TnVb5RhY8d

C:\SI\windows>SIPInspDlxCLI.exe -sn Kc4mQ9wXz2LpR7vTn1sBe6
SUCCESS

```

Command Line Interface

On Windows the executable for the command line interface is SIPInspDlxCLI.exe; on other operating systems it is SIPInspDlx when it is started with options. In the examples below the program is called SIPInspDlx. SIPInspDlx -h prints all options.

The CLI mode runs a scenario that was made in the GUI (or by hand, or by Trace to Scenario) without any window. Everything that the GUI takes from its fields is given with an option: the local and the remote address, the From and To users, the authentication, the speed of the calls and so on. The options replace the same keywords as the GUI fields do.

Option	What it does	Notes
-fs <path>	Scenario file	The scenario to run.

Option	What it does	Notes
-fv <path>	Values file	Values replace the [fieldN] keywords, one row for every new call.
-fvo	Use each row of the values file once	One call per row; the run ends after the last row, also with -tc 0.
-lip <IP address>	Local IP address	Replaces [local_ip].
-lp <port>	Local SIP port, 5060 by default	Replaces [local_port].
-rip <IP address>	Remote IP address	Where SIP messages are sent to. Replaces [remote_ip].
-rp <port>	Remote SIP port, 5060 by default	Replaces [remote_port].
-lrtp <port>	Local RTP port, 6000 by default	Replaces [media_port]. It grows by 2 for every new call.
-rtp_ip <IP address>	Remote RTP IP address	Where an unconditional RTP stream is sent (see Remote RTP).
-rtp_port <port>	Remote RTP port	Port of an unconditional RTP stream.
-t <protocol>	Transport: UDP (default), TCP, WS, WSS or TLS	Replaces [transport].
-ip <(v4 v6)>	IP version, v4 by default	Replaces [IP_ver].
-rpath <path>	Remote path for WS and WSS	The path on the remote server. Empty by default.
-uf <user>	From user name	Replaces [fromUsername].
-ut <user>	To user name	Replaces [toUsername].
-au <username>	Authentication user name	Replaces [authUsername].
-ap <password>	Authentication password	Replaces [authPassword].
-cps <float>	Call generation speed, calls per second	0.1 is one call every 10 seconds.
-mc <integer>	Maximum concurrent calls	0 means unlimited.
-tc <integer>	Total number of calls generated	0 means unlimited: the scenario runs until it is stopped.
-to <sec>	Maximum time for the scenario to complete	A scenario finishes when its end is reached. After this time it is terminated.
-noCID	Do not check Call-ID	One call gets every message, in the order of the scenario. See Don't check Call-ID.
-np	Do not print the scenario progress	Use it for a server that runs in the background, or when the output goes to a file.
-log	Log SIP messages	The messages are written to a file in the Logs folder.
-logScen	Log the scenario progress	The progress table is written to a file in the Logs folder.
-d <level>	Program log level: ERROR, WARN, DEBUG, VERBOSE or OFF	Messages go to SIPInspector.log (INFO and FINE are accepted for DEBUG and VERBOSE).
-pv	Decode the received video	Off by default: the video is then received and dropped, and Wait video does not wait. With -pv the pictures are decoded and counted.
-nv	Do not decode the received video (the default)	Video is still offered and sent.
-cTLS	Check the client's certificate	For a server that uses TLS or WSS.
-addCrt <pk> <cp>	Add own certificate (server)	pk is the private key path, cp the certificate path.
-addTCA <cp>	Add a trusted certificate or CA	cp is the certificate path.
-genKeys <a> <len>	Create a public and a private key	a is RSA, DSA, DH or EC. The keys go to the security folder.

Option	What it does	Notes
-genCrt <k1> <k2> <DN> <N>	Create a self-signed certificate	k1 private key, k2 public key, DN the domain or IP address, N the days it is valid.
-junit <file>	Write a JUnit XML report	Every scenario step is a test case; a step with an unexpected message or a failed assertion is a failure. For CI servers.
-report <file>	Write a CSV report	The run totals (calls, ASR, ACD, response times, RTP quality) and every step with PASS or FAIL.
-html <file>	Write an HTML report	A page with the verdict, the key numbers and every step. See Assertions and run reports.
-json <file>	Write the result as JSON	The verdict and why, the gate, the numbers of the run and every step, for tools that read the result.
-gate <limits>	CI gate: limits on the whole run	For example asr>=99,p95<=300,mos>=4.0. With a gate the limits alone decide the result. See The CI gate.
-api <port>	Open the control API on this port	Other programs watch and steer the run over HTTP. Without -fs the program is a service that waits for runs. See The control API.
-apiHost <address>	Address the API listens on	Default 127.0.0.1: only this computer. 0.0.0.0 opens it to the network.
-apiToken <token>	The token of the API	At least 8 characters. Without it a random token is made and printed at the start.
-apiKeep <sec>	Keep the API open after the end	So that the final result can be read; a POST /stop closes it sooner.
-diag <file.zip>	Write the diagnostics bundle	When the run ended: the settings, the scenario, the logs, the reports and the call flows in one zip, passwords masked. Without -fs and -api it is written at once. See The diagnostics bundle.
-noChecks	Do not check the messages	The SIP/SDP checks are on by default; they only warn. See SIP and SDP checks.
-pcap2scen <capture>	Convert a capture into a scenario	A command of its own, see Trace to Scenario. -pcap2scen -h lists its options.
-m	Print the Machine-ID	
-sn <SN>	Register the serial number	
-v	Print the version	
-h	Print the help	

Exit codes. The CLI mode ends with an exit code that a script can test: **0** - the run passed; **1** - the run failed (without a gate: an unexpected message, a failed assertion, a call that did not finish, a time out; with a gate: a limit of the gate); **2** - the run could not be started: the command line is wrong (an unknown option, a value or a gate that cannot be read), the socket or the scenario file could not be opened, or the program is not registered on this computer.

Metrics. At the end of a run the CLI prints the numbers of the run: the steps that passed and failed, the calls (INVITE attempts, answered calls, ASR, ACD and the final responses), the response times and the quality of the RTP that came in. A report that cannot be written is logged; it never changes the exit code. A CI job, and what a call of 3 seconds with assertions printed:

```
SIPInspDlx -lip 10.0.0.5 -rip 10.0.0.9 -fs scenarios/my_call_with_asserts.txt -tc 1 -np \
-junit results/sip.xml -report results/sip.csv -html results/sip.html
Metrics: steps 21 (21 passed, 0 failed), calls started 1, still active 0
calls: 1 attempted, 1 answered, ASR 100.0%, ACD 3 s, final responses 200 x1
response time: samples 3, min 4 ms, avg 29 ms, p50 21 ms, p95 62 ms, p99 62 ms, max 62 ms
rtp: 1 call, packets 133, lost 0 (0.00%), jitter avg 3.5 ms max 4.7 ms, MOS avg 4.40 min 4.40
sip/sdp checks: no problems in 9 messages
```

The CI gate

Without a gate a run passes only when every step went as expected, every assertion held and every call reached the end of the scenario. That is right for a functional test. In a load test a few failed calls are normal; what counts are the numbers of the whole run. A **gate** sets limits on them, and with `-gate` the limits alone decide the result and the exit code. Failed steps and calls are still in the reports, as notes.

Name	The number
asr	Answered calls / INVITE attempts, in percent
acd	Average length of the answered calls that ended, in seconds
attempts, answered	INVITE attempts, answered calls
unfinished, failedsteps	Calls that did not reach the end of the scenario; steps with unexpected messages
avg, p50, p95, p99, max	Response times (request to final response), in ms
packets, loss	RTP packets received; lost RTP packets in percent
jitter, jittermax	Average and highest RTP jitter, in ms
mos, mosmin	Average and lowest estimated MOS
problems	SIP/SDP problems the checks found in the messages (problems==0: a run passes only with clean messages)

A limit is a name, an operator (<, <=, >, >=, ==) and a number, separated by commas. A number that the run could not measure, for example the MOS when no RTP came, fails its limit: a gate that checks nothing cannot let a run pass. A gate that cannot be read stops the program with exit code 2 before any call is made. In the JUnit report every limit is a test case, and only the limits that do not hold are failures. What a call printed whose assertion (`rtp mos >= 4.6`) failed, with a gate that held:

```
SIPInspDlx -lip 10.0.0.5 -rip 10.0.0.9 -fs scenarios/my_call_with_asserts.txt -tc 1 -np -gate
asr>=99,mos>=4.0,loss<=5
Result: PASS
Gate: asr 100% >= 99 PASS, mos 4.4 >= 4 PASS, loss 0% <= 5 PASS
Not counted, the gate decides: 1 step got unexpected messages; 1 call did not finish (1 dropped by
an error),
stuck at step 16 (---- assert rtp mos >= 4.6) x1
```

The control API

This section and the next list the requests; the chapter The REST API step by step shows them at work, with what every command printed.

With `-api <port>` a CLI run opens a small web server, and other programs watch and steer the run over HTTP while it goes on: a dashboard such as Grafana charts the ASR, the response times and the MOS live, a script stops a run early when the numbers go wrong, and a test can raise or lower the call rate without starting again. Nothing has to be installed: the server is part of Java.

Request	What it does
GET /status	Running or finished, the role, the calls started and active, the rate set and measured, and after the end the result, how the run ended and the exit code
GET /metrics	The numbers of the run now, as the JSON of <code>-json</code> ; <code>?format=prometheus</code> gives them as Prometheus text
GET /report	The report of the run so far: <code>?format=html</code> (the default), <code>csv</code> , <code>junit</code> or <code>json</code>
PUT /rate	A new rate from now on: <code>{"cps": 20}</code> , <code>{"maxConcurrent": 50}</code> or both (or <code>?cps=20</code>). <code>cps 0</code> makes no new calls until the rate is set again
POST /stop	Ends the run like Ctrl-C: the reports are written as at a normal end. After the end it closes the API
GET /ui	The control page for a web browser (see The control page in a web browser); the page needs no token, every request it makes does
GET /	The list of the requests

Safe by default. The API listens on 127.0.0.1, so only programs on the same computer reach it; `-apiHost 0.0.0.0` opens it to the network. Every request must give the token, as `Authorization: Bearer <token>` or `X-API-Token: <token>`; a request without it gets 401. No request names a file. The API is opened only for a run that was started: a port that is in use stops the program with exit code 2 before any call is made. Only a client has a rate to change; a server (without `-rip`) answers `PUT /rate` with 409.

The result of a stopped run. Without a gate a run that is stopped, by Ctrl-C, by a time out or by `POST /stop`, did not finish and fails (exit code 1). With `-gate` the limits decide, so a load test without an end of its own (`-tc 0`) is stopped over the API and judged by its numbers. A run that starts at 100 calls a second, is raised by hand to 500 and then 1000, has its numbers read and is stopped:

```
SIPInspDlx -lip 10.0.0.5 -rip 10.0.0.9 -fs scenarios/Reg_client.txt -cps 100 -tc 0 -np \
    -api 8080 -apiToken s3cret-token -apiKeep 60 -gate failedsteps<=10,p95<=300 -json
run.json

curl -H "Authorization: Bearer s3cret-token" http://127.0.0.1:8080/status
curl -H "Authorization: Bearer s3cret-token" -X PUT -d '{"cps": 500}' http://127.0.0.1:8080/rate
curl -H "Authorization: Bearer s3cret-token" -X PUT -d '{"cps": 1000}' http://127.0.0.1:8080/rate
curl -H "Authorization: Bearer s3cret-token" "http://127.0.0.1:8080/metrics?format=prometheus"
curl -H "Authorization: Bearer s3cret-token" -X POST http://127.0.0.1:8080/stop
```

Prometheus reads `/metrics?format=prometheus` directly; its names are the names of the CSV report, for example `sipinspector_calls_asrPercent`, `sipinspector_responseTime_p95Ms`, `sipinspector_rtp_mosAvg`, and `sipinspector_running` is 1 while the run goes on. The bearer token goes in the authorization part of the scrape job.

The service: runs on request

Started with `-api` and without `-fs`, the program makes no run of its own: it is a **service**. It waits, and every `POST /run` makes one run. When the run is over the service goes back to the settings of its command line and waits for the next one. A lab computer then keeps one SIP Inspector running, and a script, a CI job or a web page sends it test after test: a regression suite of 40 scenarios with a JUnit report each, both ends of a call driven from one place (the answering service first, then the calling one), a test that a developer starts with one command, or the same scenario with more and more calls until the gate fails.

Request	What it does
<code>POST /run</code>	Starts a run. The body is a JSON object: <code>{"scenario": "Reg_client.txt", "remote": "10.0.0.9", "totalCalls": 100}</code> . 202 and the number of the run
<code>GET /status</code>	idle, or the run that is going on; after a run its result, how it ended, its exit code and, for a run that could not be started, why
<code>GET /runs</code>	The runs, the newest last (the last 20 are kept); <code>GET /runs/N</code> one of them with what was asked for
<code>GET /scenarios</code>	The scenario files a run can take: the .txt files of the scenarios folder of the program, with the path to give <code>POST /run</code>
<code>POST /scenarios</code>	A scenario file from another computer: <code>?name=X.txt</code> and the file as the body (at most 1 MB). It is kept as <code>scenarios/uploads/X.txt</code> (a file of that name there is replaced) and the answer gives the path to run
<code>?run=N</code>	<code>GET /metrics</code> and <code>GET /report</code> of an earlier run: <code>GET /report?run=3&format=junit</code>
<code>POST /stop</code>	Ends the run that is going on; the service takes the next one
<code>POST /shutdown</code>	Ends the program (exit code 0); a run that is going on ends first, with its reports
<code>GET /diag</code>	The diagnostics bundle (a zip) of the run now or the last one; also without the service

Name in <code>POST /run</code>	Option	Name in <code>POST /run</code>	Option
scenario	<code>-fs</code>	totalCalls	<code>-tc</code>
values, valuesOnce	<code>-fv</code> , <code>-fvo</code>	cps, maxConcurrent	<code>-cps</code> , <code>-mc</code>
remote, remotePort	<code>-rip</code> , <code>-rp</code>	timeout	<code>-to</code>

Name in POST /run	Option	Name in POST /run	Option
localIp, localPort, mediaPort	-lip, -lp, -lrtp	gate	-gate
transport, ipVersion	-t, -ip	remoteRtpIp, remoteRtpPort	-rtp_ip, -rtp_port
fromUser, toUser	-uf, -ut	wsPath	-rpath
authUser, password	-au, -ap	logSip, logScenario, processVideo	-log, -logScen, -pv
noCallIdCheck, checkClientCert, noServerNameCheck	-noCID, -cTLS, -noSNC	noChecks	-noChecks

A name is the option of the command line it stands for; a flag (valuesOnce, logSip...) is true or false. Every run starts from the settings of the command line of the service, so what they all share (the local address, the transport, the user names, a gate) is given once, at the start. A run with remote makes calls; a run without it answers them, unless the command line has -rip. **Files:** the scenario and the values file must be in the program folder; a bare name is looked for in its scenarios and values folders too. A run cannot name a report file: the reports of every run are kept by the service and read with GET /report. A wrong run (no scenario, a file outside the program folder, a name that is not known, a value that is not a number) is refused with 400 before anything is done, and a run while one is going on with 409. A scenario that cannot be loaded fails that run only (exit code 2 in its status).

```
SIPInspDlx -api 8080 -apiToken s3cret-token -lip 10.0.0.5 -t udp -au alice -ap secret
Service: waiting for runs (POST /run); POST /shutdown or Ctrl-C ends it

curl -H "Authorization: Bearer s3cret-token" -X POST http://10.0.0.5:8080/run \
  -d '{"scenario": "Reg_call_client.txt", "remote": "10.0.0.9", "totalCalls": 20, "cps": 2,
    "gate": "asr>=99"}'
{"run": 1, "state": "starting", "scenario": "/opt/si/scenarios/Reg_call_client.txt", "status":
  "/runs/1"}

curl -H "Authorization: Bearer s3cret-token" http://10.0.0.5:8080/runs/1
curl -H "Authorization: Bearer s3cret-token" "http://10.0.0.5:8080/report?run=1&format=junit" >
  call_test.xml
```

Note: A service that is reached from other computers needs -apiHost 0.0.0.0 (or the address of one network card). Keep the token secret: whoever has it can make calls from this computer.

The progress screen of the CLI

Unless -np is given, the program prints the scenario with its counters and refreshes the screen as the run goes on. The columns are the same as in the scenario screen of the GUI: the entry, how many times it was executed as expected (Exp), how many unexpected messages came while the scenario was at that entry (Unexp) and which messages those were. At the end the program prints **SUCCESS!**, or **FAILURE** and the unexpected messages.

```

Terminal - client

$ SIPInspDlx -lip 127.0.0.1 -lp 5080 -rip 127.0.0.1 -rp 5070 -fs scenarios/Reg_call_client.txt
-uf 1000 -ut 2000 -cps 2 -mc 4 -tc 8

|---> REGISTER          |          3|          0|
|<--- 200               |          3|          0|
|New Dialog             |          3|          0|
|---> INVITE            |          3|          0|
|<--- 100*              |          3|          0|
|<--- 180               |          3|          0|
|<--- 200               |          3|          0|
|---> ACK               |          3|          0|
|---- Play pcap=test.pcap |          3|          0|
|---- Pause[ms]=3000     |          3|          0|
|---> BYE               |          0|          0|
|<--- 200               |          0|          0|

```

Figure 5 - A client that makes 8 calls, two per second, at most 4 at a time. The scenario is half way through

```

Terminal - client

$ SIPInspDlx -lip 127.0.0.1 -lp 5080 -rip 127.0.0.1 -rp 5070 -fs scenarios/Reg_call_client.txt
-uf 1000 -ut 2000 -cps 2 -mc 4 -tc 8

|---> REGISTER          |          8|          0|
|<--- 200               |          8|          0|
|New Dialog             |          8|          0|
|---> INVITE            |          8|          0|
|<--- 100*              |          8|          0|
|<--- 180               |          8|          0|
|<--- 200               |          8|          0|
|---> ACK               |          8|          0|
|---- Play pcap=test.pcap |          8|          0|
|---- Pause[ms]=3000     |          8|          0|
|---> BYE               |          8|          0|
|<--- 200               |          8|          0|
SUCCESS!

```

Figure 6 - The same run when all calls are done

Note: When SIP Inspector runs as a server it has no calls to generate: it waits for the other side. Start the server first, with `-np` if it runs in the background, and then the client, as shown in the next examples.

CLI progress screen on Windows

When keys and certificates are generated and while scenarios run, the screen is updated automatically. The MS-DOS console of older Windows versions does not follow the ANSI terminal standard. For the progress screen to work well there, execute the following command once. It installs the needed DLLs and makes the console interpret ANSI codes:

```

C:\SI\windows>.\ansicon\x64\ansicon.exe -i // 64-bit Windows
C:\SI\windows>.\ansicon\x86\ansicon.exe -i // 32-bit Windows

```

Examples

A server and a client on the same computer. The server answers the registration and the call of `Reg_call_server.txt`; the client plays `Reg_call_client.txt`. The server takes up to 20 calls, so it can serve the client; `-np` keeps its output quiet:

```

SIPInspDlx -lip 127.0.0.1 -lp 5070 -lrtp 7000 -fs scenarios/Reg_call_server.txt -mc 20 -tc 20 -np

SIPInspDlx -lip 127.0.0.1 -lp 5080 -lrtp 6000 -rip 127.0.0.1 -rp 5070 \
-uf 1000 -ut 2000 -cps 2 -mc 4 -tc 8

```

The next command executes the scenario `Reg_call_client.txt`. The program binds to the local IP 192.168.1.110 and the local port 8000. Messages are sent to the remote IP 192.168.1.98 and port 60060. The From user is 6661234 and the To user 5551234. Calls are generated at 1 call per second, at most 1 call at a time, and endlessly (-tc 0), which means the scenario is executed forever:

```
SIPInspDlxCLI.exe -lip 192.168.1.110 -lp 8000 -rip 192.168.1.98 -rp 60060 -fs
scenarios\Reg_call_client.txt -uf 6661234 -ut 5551234 -cps 1 -mc 1 -tc 0
```

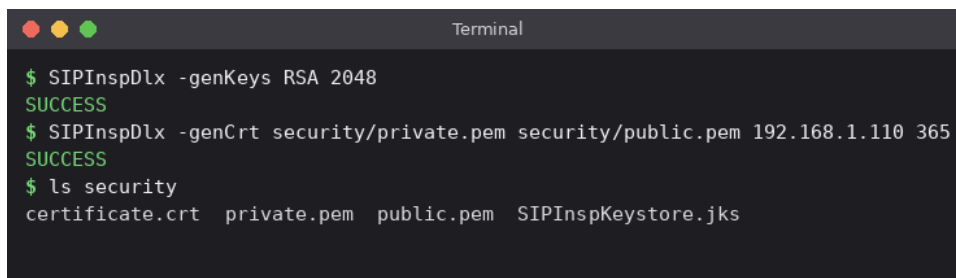
A scenario that registers many users, one for each row of a values file, 20 at a time, and logs what the program does at the DEBUG level:

```
SIPInspDlx -lip 10.0.0.5 -rip 10.0.0.1 -fs scenarios/Reg_series_client.txt -fv
values/Reg_series_values.txt \
-cps 20 -mc 20 -tc 100 -d DEBUG
```

A scenario that has the messages of several dialogs in one sequence, in the order they were captured (made with Trace to Scenario and -order capture), runs with -noCID:

```
SIPInspDlx -lip 192.168.1.110 -rip 192.168.1.98 -fs scenarios/trace_in_capture_order.txt -noCID -tc
1
```

Generate RSA keys (4096 bits) and a self-signed certificate for the IP address 192.168.1.110, valid for 365 days. The files are created in the security folder:



```
Terminal
$ SIPInspDlx -genKeys RSA 2048
SUCCESS
$ SIPInspDlx -genCrt security/private.pem security/public.pem 192.168.1.110 365
SUCCESS
$ ls security
certificate.crt private.pem public.pem SIPInspKeystore.jks
```

Figure 7 - Keys and a self-signed certificate on the command line

The certificate that SIP Inspector generates is put into its keystore at once. -addCrt is for a key and a certificate that were made by something else. A TLS server then starts like this:

```
SIPInspDlx -addCrt my_private.pem my_certificate.crt
SIPInspDlx -t TLS -lip 192.168.1.110 -lp 5061 -fs scenarios/Reg_server.txt -mc 20 -tc 0 -np
```

The REST API step by step

This chapter shows the control API at work: first one run that is watched, sped up and stopped, then the service that takes run after run. Every picture is what the commands printed. The examples use `curl`, which is part of Linux, macOS and Windows 10 and later; any program that speaks HTTP does the same. The requests and their answers are listed in The control API and The service: runs on request.

Every request needs the token. The examples keep the header in a shell variable, so the commands stay short:

```
H="Authorization: Bearer s3cret-token"
curl -s -H "$H" http://10.0.0.5:8080/status           (Linux, macOS)

curl.exe -s -H "Authorization: Bearer s3cret-token" http://10.0.0.5:8080/status           (Windows)
```

The control page in a web browser

The API has a page of its own for a web browser: `http://<address>:<port>/ui`, for example `http://10.0.0.5:8080/ui`. The program prints the address at the start (Control page: ...). Nothing is installed and nothing comes from the internet: the page is part of the program, so it works in a lab network too. It asks for the token once:

Figure 8 - The control page asks for the token

Then it shows the run as it goes on, refreshed every second: the calls, the measured rate, the ASR, the p95 response time, the MOS and the SIP/SDP problems, a chart of the calls over time, the gate, the steps of the scenario, the checks and the call flow of a call. It has what the requests of the API do, as buttons:

- **Call rate** - a new rate or limit of calls at a time, from now on; Pause sets 0 calls a second (a client only).
- **Stop run** - ends the run like Ctrl-C, with its reports. **Shut down the program** ends the program.
- **Reports** - the HTML report opens in a new tab; CSV, JUnit XML, JSON and the diagnostics bundle are saved.
- **Start a run** and **Runs** (the service) - the form of POST /run: the scenario, the remote end, the calls, the rate, a gate and, under More names, any other name of a run as JSON. The runs are listed with their result and their reports.
- **Browse...** next to the scenario lists the scenario files on the computer SIP Inspector runs on, with a filter; a click puts one in the form. **Upload...** sends a scenario file of your own computer: it is kept in scenarios/uploads there and put in the form.

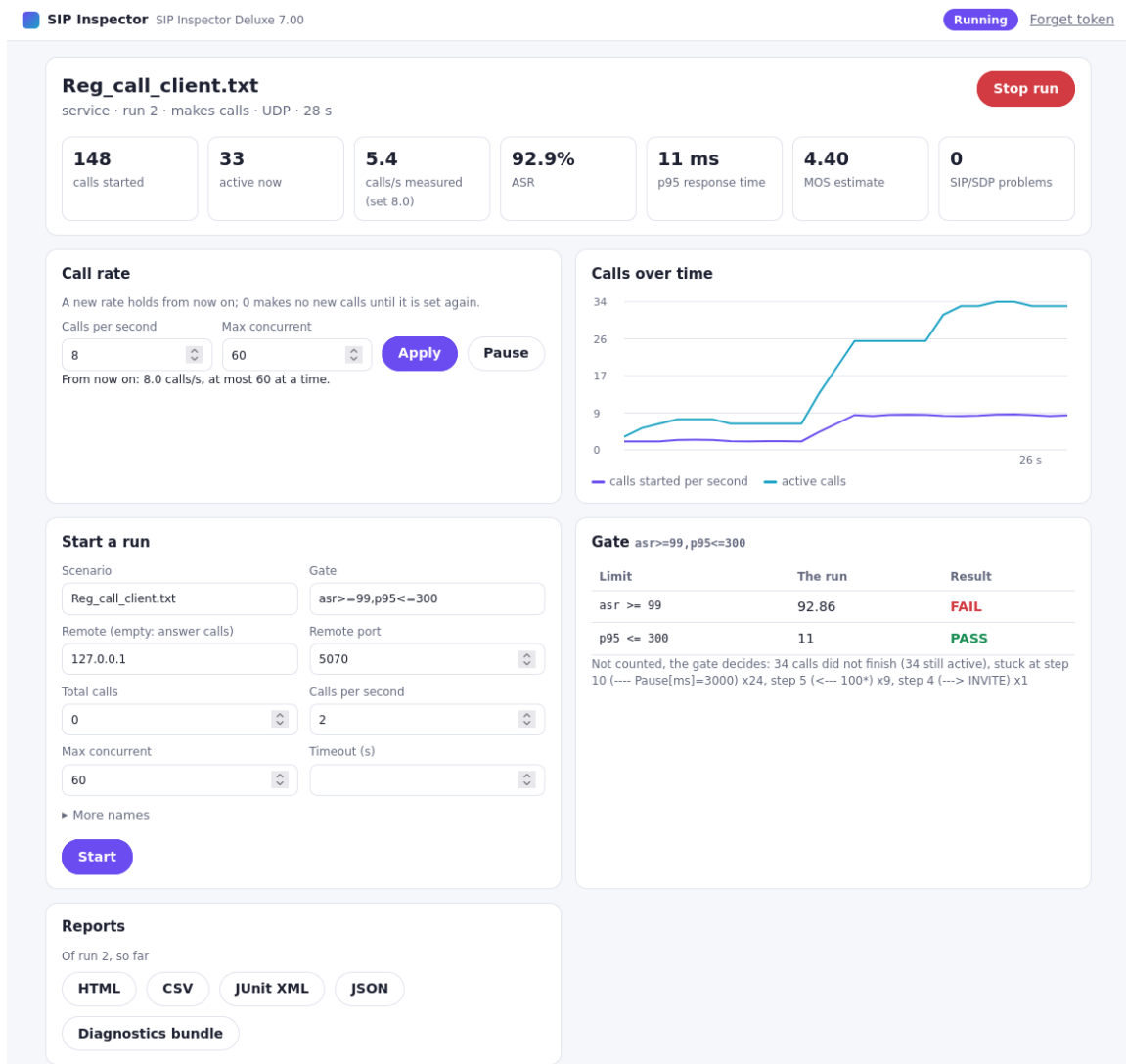


Figure 9 - The control page of a service while its second run goes on: the rate was raised from 2 to 8 calls a second from the page

Runs							
#	Scenario	Started	Seconds	Calls	Result	Exit	Reports
2	Reg_call_client.txt	2026-10-16 19:41:23	29.6	159	FAIL	1	HTML JUnit CSV
1	Reg_call_client.txt	2026-10-16 19:41:13	6.7	6	PASS	0	HTML JUnit CSV

Figure 10 - The runs of the service, each with its reports

The token. The page keeps it only while its tab is open, unless **Remember it in this browser** is ticked; **Forget token** removes it. Every request the page makes gives it, like curl does; the page itself has nothing secret and comes without a token. A page of another web site cannot use the API through your browser: the API answers it with 401, as it has no token. The page follows the light or dark theme of the computer, and fits a phone too.

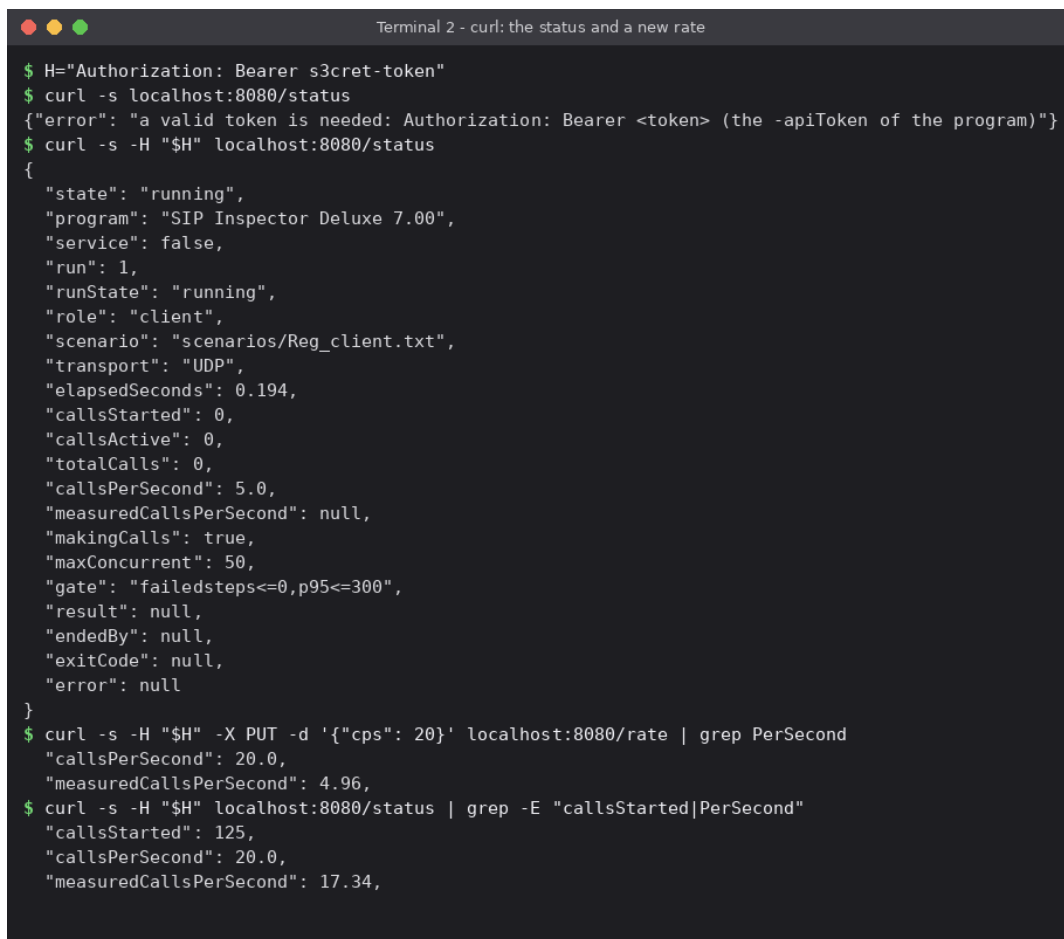
1. A run with the API

The client registers users, 5 a second and with no end of its own (-tc 0), against a server on port 5070. -api 8080 opens the API, -apiToken sets the token (without it the program makes one and prints it), -apiKeep 30 keeps the API open 30 seconds after the end so the result can still be read, and the gate decides the result. The program prints where the API is, then works as usual:

```
SIPInspDlx -fs scenarios/Reg_client.txt -lp 127.0.0.1 -lp 5080 -rip 127.0.0.1 -rp 5070 -t udp -cps
5 -tc 0 -np \
    -api 8080 -apiToken s3cret-token -apiKeep 30 -gate "failedsteps<=0,p95<=300"
API: http://127.0.0.1:8080/
Control page: http://127.0.0.1:8080/ui
```

2. Watch it and change the rate

From another terminal, or another computer when the API listens there (-apiHost). A request without the token gets 401. **GET /status** says what the run does now. **PUT /rate** sets a new rate from now on and answers with the status; a few seconds later the measured rate has followed:

A terminal window titled "Terminal 2 - curl: the status and a new rate" showing a series of curl commands and their JSON outputs. The first command shows an error for missing token. The second shows the current status with a rate of 5.0 calls per second. The third command uses PUT to change the rate to 20.0. The final status shows the rate has been updated to 20.0 and the measured rate is 17.34.

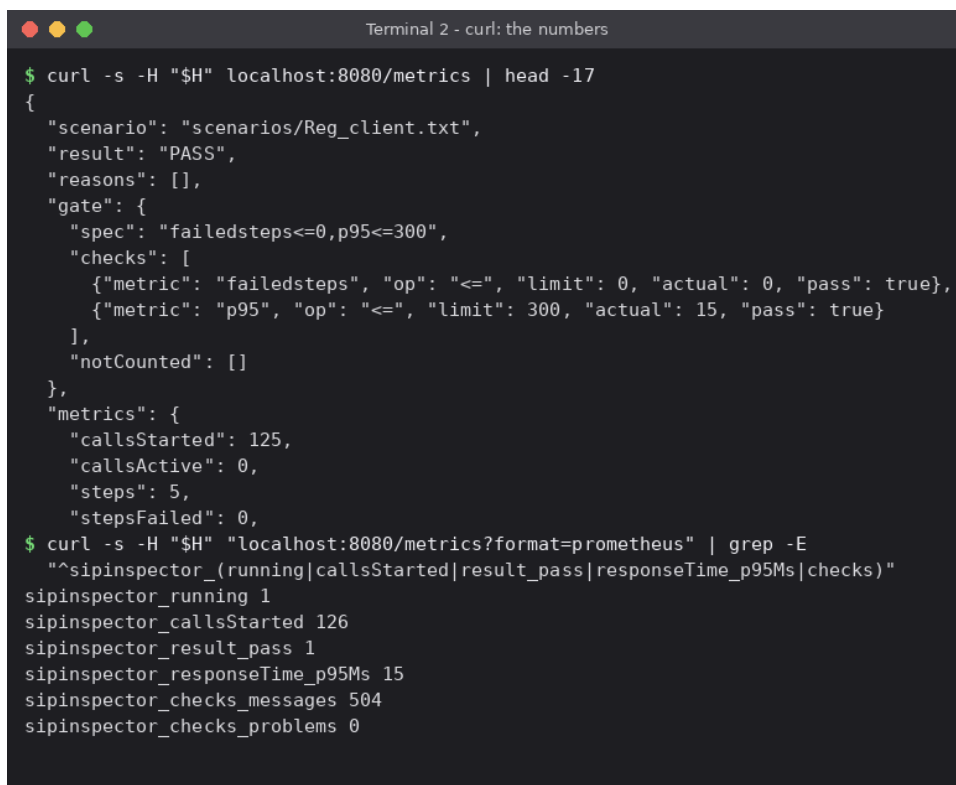
```
$ H="Authorization: Bearer s3cret-token"
$ curl -s localhost:8080/status
{"error": "a valid token is needed: Authorization: Bearer <token> (the -apiToken of the program)"}
$ curl -s -H "$H" localhost:8080/status
{
  "state": "running",
  "program": "SIP Inspector Deluxe 7.00",
  "service": false,
  "run": 1,
  "runState": "running",
  "role": "client",
  "scenario": "scenarios/Reg_client.txt",
  "transport": "UDP",
  "elapsedSeconds": 0.194,
  "callsStarted": 0,
  "callsActive": 0,
  "totalCalls": 0,
  "callsPerSecond": 5.0,
  "measuredCallsPerSecond": null,
  "makingCalls": true,
  "maxConcurrent": 50,
  "gate": "failedsteps<=0,p95<=300",
  "result": null,
  "endedBy": null,
  "exitCode": null,
  "error": null
}
$ curl -s -H "$H" -X PUT -d '{"cps": 20}' localhost:8080/rate | grep PerSecond
"callsPerSecond": 20.0,
"measuredCallsPerSecond": 4.96,
$ curl -s -H "$H" localhost:8080/status | grep -E "callsStarted|PerSecond"
"callsStarted": 125,
"callsPerSecond": 20.0,
"measuredCallsPerSecond": 17.34,
```

Figure 11 - The status of the run, and the rate raised from 5 to 20 calls a second

In the status	What it is
state	idle (the service between runs), running or finished
program	The name and the version of the program
run, runState	The number of the run, and its state: starting, running, finished, or failed (a run that could not be started)
role, scenario, transport	client (makes calls) or server (answers them), the scenario file and the transport
elapsedSeconds	How long the run has been going
callsStarted, callsActive, totalCalls	Calls started so far, calls going on now, and the end (-tc; 0: no end)
callsPerSecond, measuredCallsPerSecond	The rate that is set, and the rate measured since the last change
makingCalls, maxConcurrent	Whether new calls are made now (false at cps 0 or after the last one), and the limit of calls at a time
gate	The limits of -gate, or null
result, endedBy, exitCode	After the end: PASS or FAIL, how the run ended (all calls done, stopped by the API, a signal, the timeout) and its exit code
error	Why a run could not be started (the service), or null

3. The numbers

GET /metrics gives the numbers of the run so far as the JSON report: the verdict, the gate with the number of every limit, and the totals. With `?format=prometheus` they are Prometheus text, one number a line:

A terminal window titled "Terminal 2 - curl: the numbers" shows two curl commands and their outputs. The first command is `curl -s -H "$H" localhost:8080/metrics | head -17`, which returns a JSON object. The second command is `curl -s -H "$H" "localhost:8080/metrics?format=prometheus" | grep -E "^sipinspector_(running|callsStarted|result_pass|responseTime_p95Ms|checks)"`, which returns Prometheus text metrics.

```
$ curl -s -H "$H" localhost:8080/metrics | head -17
{
  "scenario": "scenarios/Reg_client.txt",
  "result": "PASS",
  "reasons": [],
  "gate": {
    "spec": "failedsteps<=0,p95<=300",
    "checks": [
      {"metric": "failedsteps", "op": "<=", "limit": 0, "actual": 0, "pass": true},
      {"metric": "p95", "op": "<=", "limit": 300, "actual": 15, "pass": true}
    ],
    "notCounted": []
  },
  "metrics": {
    "callsStarted": 125,
    "callsActive": 0,
    "steps": 5,
    "stepsFailed": 0,
$ curl -s -H "$H" "localhost:8080/metrics?format=prometheus" | grep -E
^sipinspector_(running|callsStarted|result_pass|responseTime_p95Ms|checks)"
sipinspector_running 1
sipinspector_callsStarted 126
sipinspector_result_pass 1
sipinspector_responseTime_p95Ms 15
sipinspector_checks_messages 504
sipinspector_checks_problems 0
```

Figure 12 - The numbers of the run as JSON (the start of it) and as Prometheus text (a few of the lines)

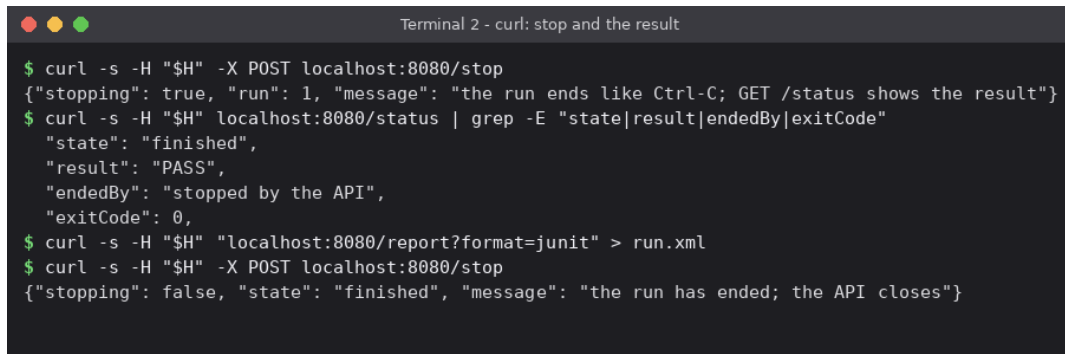
Prometheus reads them directly. A scrape job, every 5 seconds, with the token:

```
scrape_configs:
- job_name: sipinspector
  scrape_interval: 5s
  metrics_path: /metrics
  params:
    format: [prometheus]
  authorization:
    credentials: s3cret-token
  static_configs:
    - targets: ["10.0.0.5:8080"]
```

A Grafana panel then charts, for example, `sipinspector_responseTime_p95Ms`, `sipinspector_calls_asrPercent`, `sipinspector_rtp_mosAvg` or `rate(sipinspector_callsStarted[1m])`; `sipinspector_running` is 1 while the run goes on and `sipinspector_result_pass` is 1 while the result would be PASS.

4. Stop it and read the result

POST /stop ends the run like Ctrl-C: the calls that are going on are stopped, and the reports are written as at a normal end. The status then has the result and how the run ended, and **GET /report** gives the report in any format. Because of `-apiKeep` the API stays open; a second **POST /stop** closes it at once.



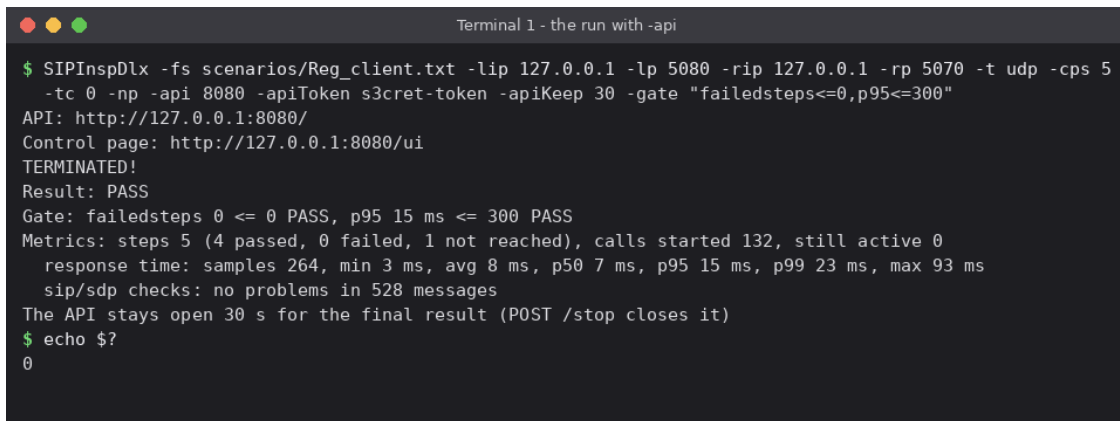
```

$ curl -s -H "$H" -X POST localhost:8080/stop
{"stopping": true, "run": 1, "message": "the run ends like Ctrl-C; GET /status shows the result"}
$ curl -s -H "$H" localhost:8080/status | grep -E "state|result|endedBy|exitCode"
"state": "finished",
"result": "PASS",
"endedBy": "stopped by the API",
"exitCode": 0,
$ curl -s -H "$H" "localhost:8080/report?format=junit" > run.xml
$ curl -s -H "$H" -X POST localhost:8080/stop
{"stopping": false, "state": "finished", "message": "the run has ended; the API closes"}

```

Figure 13 - Stopped by the API: the result, and the JUnit report saved

The run itself printed its summary and ended. With the gate the limits decide, so a run without an end of its own that is stopped over the API passes when its numbers are good (exit code 0):



```

$ SIPInspDlx -fs scenarios/Reg_client.txt -lip 127.0.0.1 -lp 5080 -rip 127.0.0.1 -rp 5070 -t udp -cps 5
-tc 0 -np -api 8080 -apiToken s3cret-token -apiKeep 30 -gate "failedsteps<=0,p95<=300"
API: http://127.0.0.1:8080/
Control page: http://127.0.0.1:8080/ui
TERMINATED!
Result: PASS
Gate: failedsteps 0 <= 0 PASS, p95 15 ms <= 300 PASS
Metrics: steps 5 (4 passed, 0 failed, 1 not reached), calls started 132, still active 0
response time: samples 264, min 3 ms, avg 8 ms, p50 7 ms, p95 15 ms, p99 23 ms, max 93 ms
sip/sdp checks: no problems in 528 messages
The API stays open 30 s for the final result (POST /stop closes it)
$ echo $?
0

```

Figure 14 - Terminal 1: the run, from the start to the end

5. The service

The service makes no run of its own: it waits for POST /run. A test lab keeps two of them, one that answers the calls (on the device side, or in place of a device) and one that makes them, and a script drives both. What both share is on their command line; what changes from test to test is in the runs. Here both are on one computer, on ports 8081 (answers) and 8082 (calls):

```

SIPInspDlx -api 8081 -apiToken s3cret-token -lip 127.0.0.1 -lp 5070 -t udp
SIPInspDlx -api 8082 -apiToken s3cret-token -lip 127.0.0.1 -lp 5080 -t udp -au alice -ap secret

```

The answering service gets a run without remote, so it answers, for at most 60 seconds. The calling one gets 6 calls with a gate. Each answers 202 with the number of the run at once; the run goes on in the background. A second run while one is going on gets 409:

```
curl: two runs started

$ H="Authorization: Bearer s3cret-token"
$ curl -s -H "$H" -X POST -d '{"scenario": "Reg_call_server.txt", "timeout": 60}' localhost:8081/run
{"run": 1, "state": "starting", "scenario": "/opt/si/scenarios/Reg_call_server.txt", "status":
"/runs/1"}
$ curl -s -H "$H" -X POST -d '{"scenario": "Reg_call_client.txt",
  "remote": "127.0.0.1", "remotePort": 5070, "totalCalls": 6, "cps": 2,
  "gate": "asr>=99,mos>=4.0"}' localhost:8082/run
{"run": 1, "state": "starting", "scenario": "/opt/si/scenarios/Reg_call_client.txt", "status":
"/runs/1"}
$ curl -s -H "$H" -X POST -d '{"scenario": "Reg_client.txt"}' localhost:8082/run
{"error": "run 1 is going on: one run at a time (POST /stop ends it)"}
$ curl -s -H "$H" localhost:8082/status | grep -E '(state|run|runState|scenario|callsStarted)'"
"state": "running",
"run": 1,
"runState": "running",
"scenario": "/opt/si/scenarios/Reg_call_client.txt",
"callsStarted": 0,
```

Figure 15 - Two runs started, one on each service

GET /runs lists the runs (the last 20 are kept), and **GET /runs/N** one of them with what was asked for. The reports of every run stay: **?run=N** picks one. The answering run, which has no end of its own, is stopped:

```
curl: the runs and a report

$ curl -s -H "$H" localhost:8082/runs
{
  "runs": [
    {
      "run": 1,
      "state": "finished",
      "scenario": "/opt/si/scenarios/Reg_call_client.txt",
      "started": "2026-10-16T19:43:18",
      "seconds": 6.705,
      "callsStarted": 6,
      "result": "PASS",
      "endedBy": "all calls done",
      "exitCode": 0,
      "error": null
    }
  ]
}
$ curl -s -H "$H" "localhost:8082/report?run=1&format=junit" > call_test.xml
$ curl -s -H "$H" -X POST localhost:8081/stop
{"stopping": true, "run": 1, "message": "the run ends like Ctrl-C; GET /status shows the result"}
```

Figure 16 - The run is over: its result, its JUnit report, and the answering run stopped

A run that is wrong is refused before anything is done, with 400 and the reason: a file outside the program folder, a name that is not a name of a run, a value that is not a number. **GET /diag** gives the diagnostics bundle, and **POST /shutdown** ends a service:

```

curl: wrong runs, the bundle and the end

$ curl -s -H "$H" -X POST -d '{"scenario": "../../../etc/passwd"}' localhost:8082/run
{"error": "../../../etc/passwd is not in the program folder /opt/si: a run takes only files from there"}
$ curl -s -H "$H" -X POST -d '{"scenario": "Reg_client.txt", "totalCals": 5}' localhost:8082/run
{"error": "'totalCals' is not a name of a run; they are scenario, values, valuesOnce, remote,
remotePort, localIp, localPort, mediaPort, transport, ipVersion, fromUser, toUser, authUser, password,
totalCalls, cps, maxConcurrent, timeout, gate, remoteRtpIp, remoteRtpPort, wsPath, noCallIdCheck,
checkClientCert, noServerNameCheck, logSip, logScenario, processVideo, noChecks"}
$ curl -s -H "$H" -X POST -d '{"scenario": "Reg_client.txt", "cps": "fast"}' localhost:8082/run
{"error": "cps is a number of 0 or more, not fast"}
$ curl -s -H "$H" localhost:8082/runs/7
{"error": "no run 7; GET /runs lists the runs that are kept (the last 20)"}
$ curl -s -H "$H" -o diag.zip localhost:8082/diag && unzip -Zl diag.zip
about.txt
settings.txt
scenario-check.txt
logs/SIPInspector.log
reports/Reg_auth_call_client_2026-10-16_19-02-40.html
run/run.json
run/run.html
run/checks.txt
run/callflows.txt
README.txt
$ curl -s -H "$H" -X POST localhost:8082/shutdown
{"shuttingDown": true, "message": "a run that is going on ends first, with its reports"}
$ curl -s -H "$H" -X POST localhost:8081/shutdown
{"shuttingDown": true, "message": "a run that is going on ends first, with its reports"}

```

Figure 17 - Wrong runs, the diagnostics bundle, and the end of both services

The calling service printed the summary of its run, and ended with exit code 0:

```

The calling service

$ SIPInspDlx -api 8082 -apiToken s3cret-token -lip 127.0.0.1 -lp 5080 -t udp -au alice -ap secret
API: http://127.0.0.1:8082/
Control page: http://127.0.0.1:8082/ui
Service: waiting for runs (POST /run); POST /shutdown or Ctrl-C ends it
SUCCESS!
Result: PASS
Gate: asr 100% >= 99 PASS, mos 4.4 >= 4 PASS
Metrics: steps 12 (12 passed, 0 failed), calls started 6, still active 0
  calls: 6 attempted, 6 answered, ASR 100.0%, ACD 3 s, final responses 200 x6
  response time: samples 18, min 8 ms, avg 20 ms, p50 12 ms, p95 85 ms, p99 85 ms, max 85 ms
  rtp: 4 calls, packets 866, lost 0 (0.00%), jitter avg 0.0 ms max 0.4 ms, MOS avg 4.40 min 4.40
  sip/sdp checks: no problems in 54 messages
Service ended
$ echo $?
0

```

Figure 18 - The calling service, from the start to POST /shutdown

Answers

Code	When
200	The answer: a status, numbers, a report, a list of runs, a bundle
202	POST /run, POST /stop, POST /shutdown: taken, it happens in the background (GET /status follows it)
400	A request that is wrong: a body that is not JSON, a name that is not known, a value that is not a number, a file outside the program folder, a gate that cannot be read
401	No token, or a wrong one
404	No such request, or no such run (GET /runs/7 when 7 is not kept)
405	The request with the wrong method, for example GET /stop; the Allow header says which one
409	Not now: a run while one is going on, POST /run to a program started with -fs, PUT /rate for a server, before the first call or after the last one, POST /stop of a service without a run, the numbers of a run that could not be started
500	The program failed; the reason is in the answer and in SIPInspector.log

Every error answer is JSON with an error that says what is wrong, so a script can print it as it is.

A test from a CI job

A shell step that runs one test on the calling service, waits for its end, keeps its JUnit report and ends with the exit code of the run, so the job fails when the test fails. The token comes from a secret variable of the CI server (SI_TOKEN):

```
API=http://10.0.0.5:8082
H="Authorization: Bearer $SI_TOKEN"
RUN=$(curl -s -H "$H" -X POST -d '{"scenario": "Reg_call_client.txt", "remote": "10.0.0.9",
    "totalCalls": 20, "cps": 2, "gate": "asr>=99,mos>=4.0"}' $API/run \
    | sed -n 's/.*"run": \([0-9]*\)*/\1/p')
[ -n "$RUN" ] || exit 2
while curl -s -H "$H" $API/runs/$RUN | grep -qE '"state": "(starting|running)"; do sleep 2; done
curl -s -H "$H" "$API/report?run=$RUN&format=junit" > sip-test.xml
exit $(curl -s -H "$H" $API/runs/$RUN | sed -n 's/.*"exitCode": \([0-9]*\)*/\1/p')
```

A regression suite in Python

`samples/api/si_suite.py` runs a list of tests on the calling service, one after another, saves the JUnit report of every test and ends with exit code 1 when one failed. It needs Python 3 and nothing else. Set API, TOKEN and REMOTE at the top, and the tests in TESTS: each is a POST /run, so every name of a run can be used.

```
#!/usr/bin/env python3
"""A regression suite on a SIP Inspector service, with a JUnit report of every test."""
import json, sys, time, urllib.error, urllib.request

# the calling service: SIPInspDlx -api 8082 -apiToken s3cret-token ...
API = "http://127.0.0.1:8082"
TOKEN = "s3cret-token"
REMOTE = {"remote": "127.0.0.1", "remotePort": 5070} # the device under test

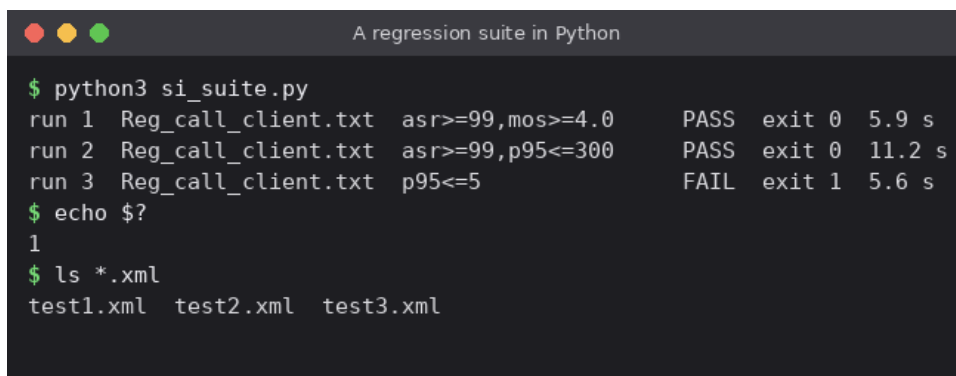
TESTS = [
    {"scenario": "Reg_call_client.txt", "totalCalls": 4, "cps": 2,
     "gate": "asr>=99,mos>=4.0"},
    {"scenario": "Reg_call_client.txt", "totalCalls": 9, "cps": 3, "maxConcurrent": 3,
     "gate": "asr>=99,p95<=300"},
    {"scenario": "Reg_call_client.txt", "totalCalls": 4, "cps": 2, "gate": "p95<=5"},
]

def call(method, path, body=None, raw=False):
    data = None if body is None else json.dumps(body).encode()
    req = urllib.request.Request(API + path, method=method, data=data,
                                 headers={"Authorization": "Bearer " + TOKEN})
    try:
        with urllib.request.urlopen(req) as r:
            text = r.read().decode()
    except urllib.error.HTTPError as e: # 400, 404, 409: the answer says why
        text = e.read().decode()
    return text if raw else json.loads(text)

failed = 0
for test in TESTS:
    run = call("POST", "/run", dict(test, **REMOTE))
    if "error" in run:
        print("not started:", run["error"])
        failed += 1
        continue
    n = run["run"]
    while True: # one run at a time: wait for its end
        status = call("GET", "/runs/%d" % n)
        if status["state"] in ("finished", "failed"):
            break
        time.sleep(1)
    with open("test%d.xml" % n, "w") as f:
        f.write(call("GET", "/report?run=%d&format=junit" % n, raw=True))
    print("run %d %-20s %-20s %s exit %s %.1f s" % (n, test["scenario"], test.get("gate", ""),
                                                    status["result"], status["exitCode"],
                                                    status["seconds"]))

    failed += status["exitCode"] != 0
sys.exit(1 if failed else 0)
```

The third test asks for a p95 of at most 5 ms, which the network does not give, so it fails and so does the suite:



```
A regression suite in Python

$ python3 si_suite.py
run 1 Reg_call_client.txt asr>=99,mos>=4.0 PASS exit 0 5.9 s
run 2 Reg_call_client.txt asr>=99,p95<=300 PASS exit 0 11.2 s
run 3 Reg_call_client.txt p95<=5 FAIL exit 1 5.6 s
$ echo $?
1
$ ls *.xml
test1.xml test2.xml test3.xml
```

Figure 19 - The suite: two tests passed, one failed, and a JUnit report of each

Keeping it safe

- The API listens on 127.0.0.1 unless -apiHost says otherwise. Open it to the network only where it is needed, for example -apiHost 10.0.0.5 (one network card) rather than 0.0.0.0.
- Whoever has the token can make calls from this computer. Use a long random token, keep it in the secrets of the CI server, and give every service its own.
- The API speaks HTTP, not HTTPS: the token can be read on the network. Keep it in a lab network, or put a reverse proxy with TLS (nginx, Caddy) in front of it.
- A run takes files only from the program folder and cannot name a file the program writes; the reports are read over the API.

Initial screen

The initial screen offers the basic account settings, the choice of the local network interface, the signalling port, the IP version, the transport protocol and the remote server. While SIP Inspector can work with values that are written into a scenario file, the best results and portability are achieved when scenarios are written with keywords. Such scenarios are highly portable between computers and between users. Keywords in a scenario are replaced with the content of certain fields of this screen. For example, if a scenario is written as if the call is made from [fromUsername] and to [toUsername], changing 'From' and 'To' in the Account info is the way to change the numbers of the caller and the called party.

SIP Inspector Deluxe 7.00

File Edit Tools Options Help

SIP Inspector Deluxe
Version 7.00 · SIP · RTP · Video

1 Account

From: 1000 Auth username: 1000

To: 2000 Auth password: secret1000

RTP Port: 6000

License Info

Machine ID: 7Gx2KqPz9Lw3TnVb5RhY8d

Serial Number: Kc4mQ9wXz2LpR7vTn1sBe6

2 Local Info

Select local IP address: 192.168.2.10

Local port: 5060

IP Version: ☒ IPv4 ☐ IPv6

Protocol: ☒ UDP ☐ TCP ☐ WS ☐ WSS ☐ TLS

3 Remote Server

☒ Remote Server

Remote Server: 192.168.1.98

Remote Server Port: 5060

Remote Path: /ws

4 Next

Figure 20 - Initial screen: (1) account, (2) local info, (3) remote server, (4) Next

1st step: Account Info provides the basic information that is used when brand new SIP messages are created. The content of 'From:' and 'To:' replaces the keywords [fromUsername] and [toUsername]. 'Auth username' and 'Auth password' replace [authUsername] and [authPassword]. The RTP port is the port the tool uses to accept and to send RTP from. It replaces the keyword [media_port]. The first call uses the given port; if more calls are generated the port grows with every new call.

2nd step: Local Info. SIP Inspector detects the installed and enabled network adapters of the computer and lists all local IP addresses. Most users have a single adapter and a single address. The SIP signalling port is typically 5060, but it can be changed. For the transport protocol choose one of UDP, TCP, WS, WSS or TLS; standard SIP uses UDP. IPv4 and IPv6 are both supported. If you are not sure what the difference is, stay with the defaults, IPv4 and UDP. For TCP and TLS the program binds to the given port, and if it works as a client an ephemeral port is used for outgoing messages.

3rd step: Remote Server. If the tool works as a client, 'Remote Server' and 'Remote Server Port' must be filled in. This is where SIP messages are sent. The field can have an IP address, an FQDN (Fully Qualified Domain Name) or an SRV name (_sip._udp.example.com). If the tool acts as a server, do not tick the 'Remote Server' check box. 'Remote Path' is used for WS and WSS.

4th step: 'Next' goes to the screen where SIP messages and scenarios are created.

Already at this screen it is possible to:

1. load a scenario file (File/Load Scenario);
2. load a values file (File/Load Values). If you ever have to deal with a variety of users, invest some energy into understanding this feature;
3. create public and private cryptographic keys (Tools/Security Keys);
4. create a self-signed certificate (Tools/Security Certificate);
5. turn a Wireshark capture into a scenario (Tools/Trace to Scenario);
6. choose the dark or the light theme and set the logging options (Options).

Scenario building screen

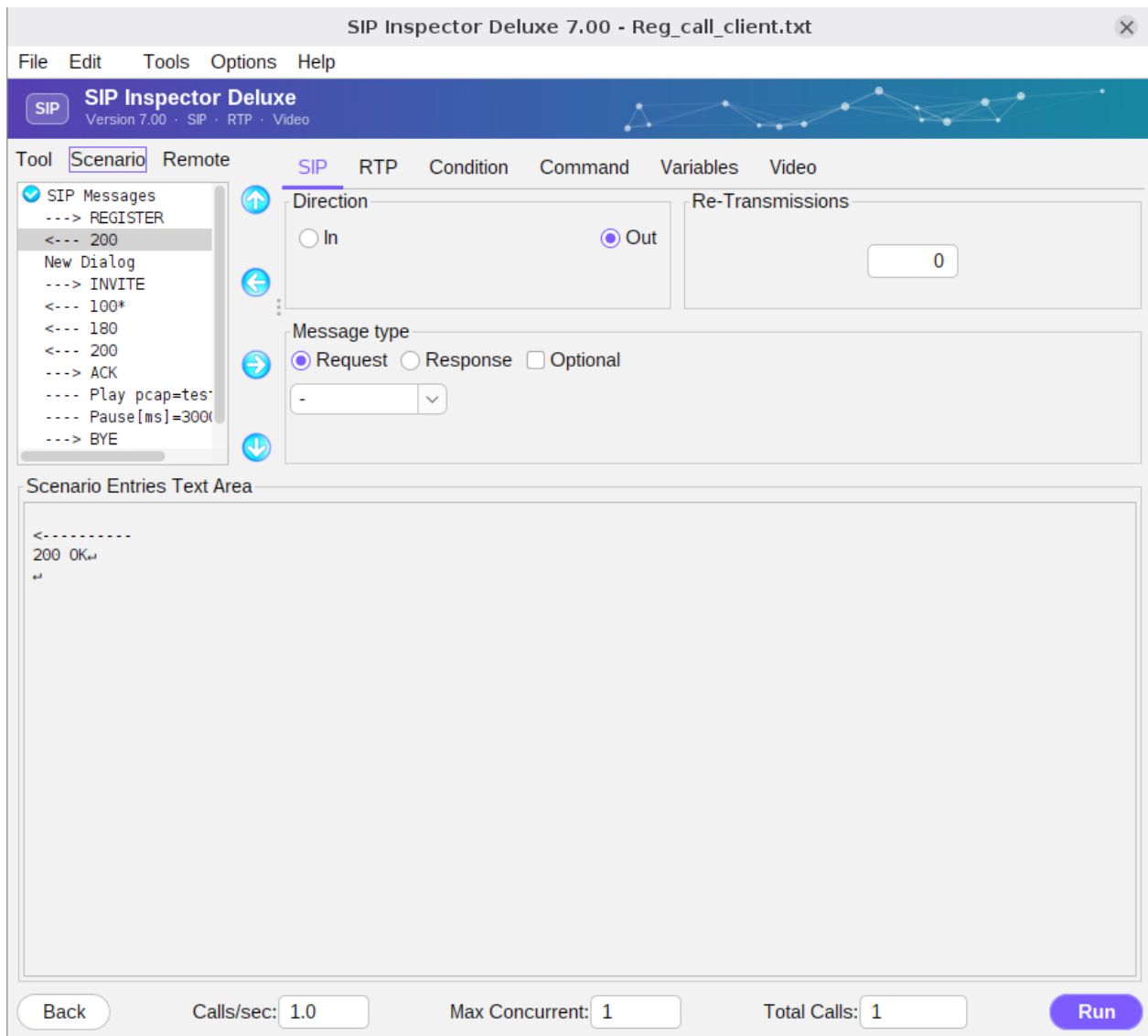


Figure 21 - Scenario building screen with a scenario loaded

The scenario building screen has 3 sections:

1. **Scenario entries.** This part of the GUI gives an overview of all the entries of the scenario. One or more entries can be selected. The content of a selected entry is displayed in the 'Scenario Entries Text Area'. Entries are added and removed with the left pointing and the right pointing arrow icon. To add an entry at a particular position select the entry that it must follow, pick the item in the scenario creation tools and press the left pointing arrow: the new entry appears below the selected one. To delete entries select them and press the right pointing arrow. The up and down arrow buttons move the selected entries. The right edge of the tree can be dragged to make the tree wider.
2. **Scenario creation tools.** Used to build a custom scenario. There are 6 tabs (a 7th, TLS, appears when the transport is TLS or WSS), split by what they offer:
 1. **SIP** - SIP signalling messages;
 2. **RTP** - entries that send live speech, play audio from a pcap file, echo the incoming RTP or send DTMF digits;
 3. **Condition** - labels, goto and conditional branching, and wait for RTP;
 4. **Command** - a pause, a remote target, a new dialog, a special command, wait for RTP, wait for video and set variable;
 5. **Variables** - custom variables. The content of a variable is usually extracted from a received message with a 'star-matching' pattern. A variable can also get a random value from a range;
 6. **Video** - shows the received video and adds an entry that waits for pictures;

7. **TLS** - the certificate of the program and, for a server, whether to ask for the client's certificate. SIP Inspector trusts self-signed certificates of a server by default, so it is not required to add the remote server's certificate.

3. Scenario Entries Text Area. Shows the text of the selected entry. In this area entries can be modified by hand, and new ones can be typed. Comment lines, which start with //, are kept when the scenario is saved. The ↵ sign marks the end of a line (CR LF) of a SIP message; it is only shown on the screen and is not part of the message. Keywords such as [local_ip] and [local_port] are visible in the text; the full list is at the end of this manual.

At the bottom of the screen: **Back** returns to the initial screen. **Calls/sec** is the speed at which calls are generated; the value can be a fraction: 0.1 generates one call every 10 seconds, 2 generates two calls every second. **Max Concurrent** limits the number of simultaneous calls (0 means unlimited). **Total Calls** tells when call generation stops (0 means endless). **Run** starts the scenario and brings up the scenario progress screen. When the program works as a server these three fields are hidden: the other side makes the calls.

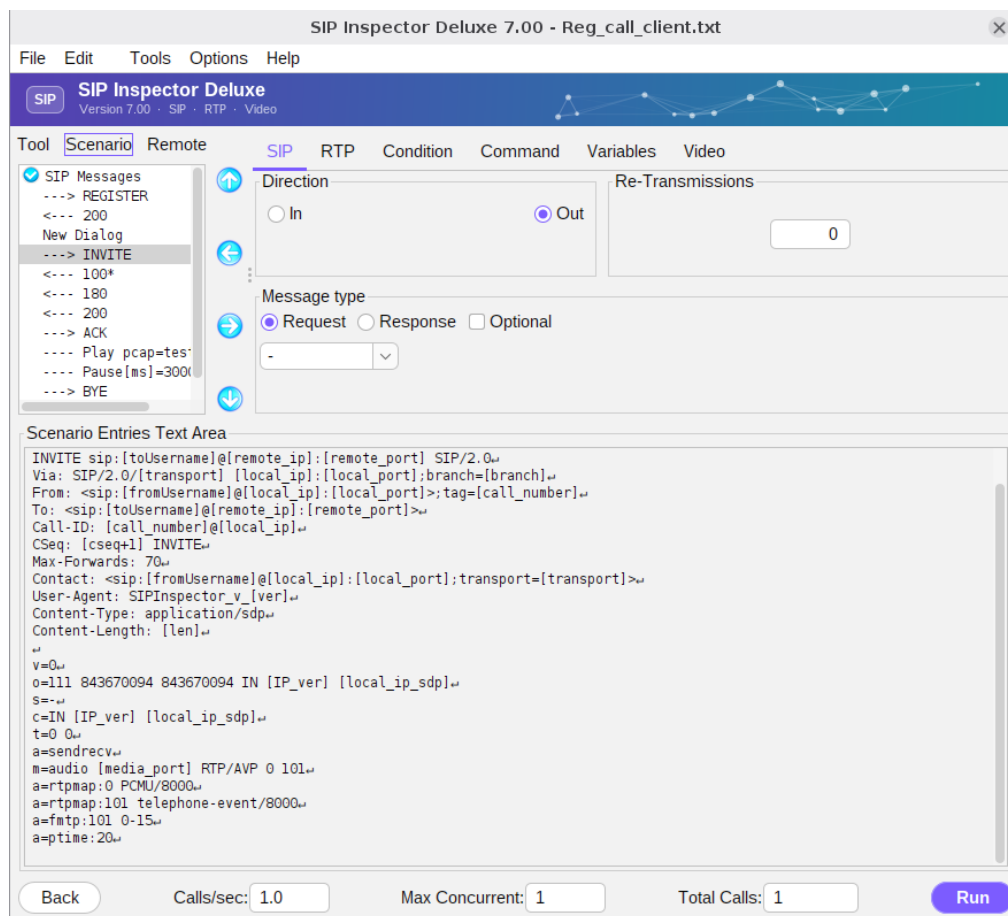


Figure 22 - The same screen with the INVITE entry selected

SIP tab

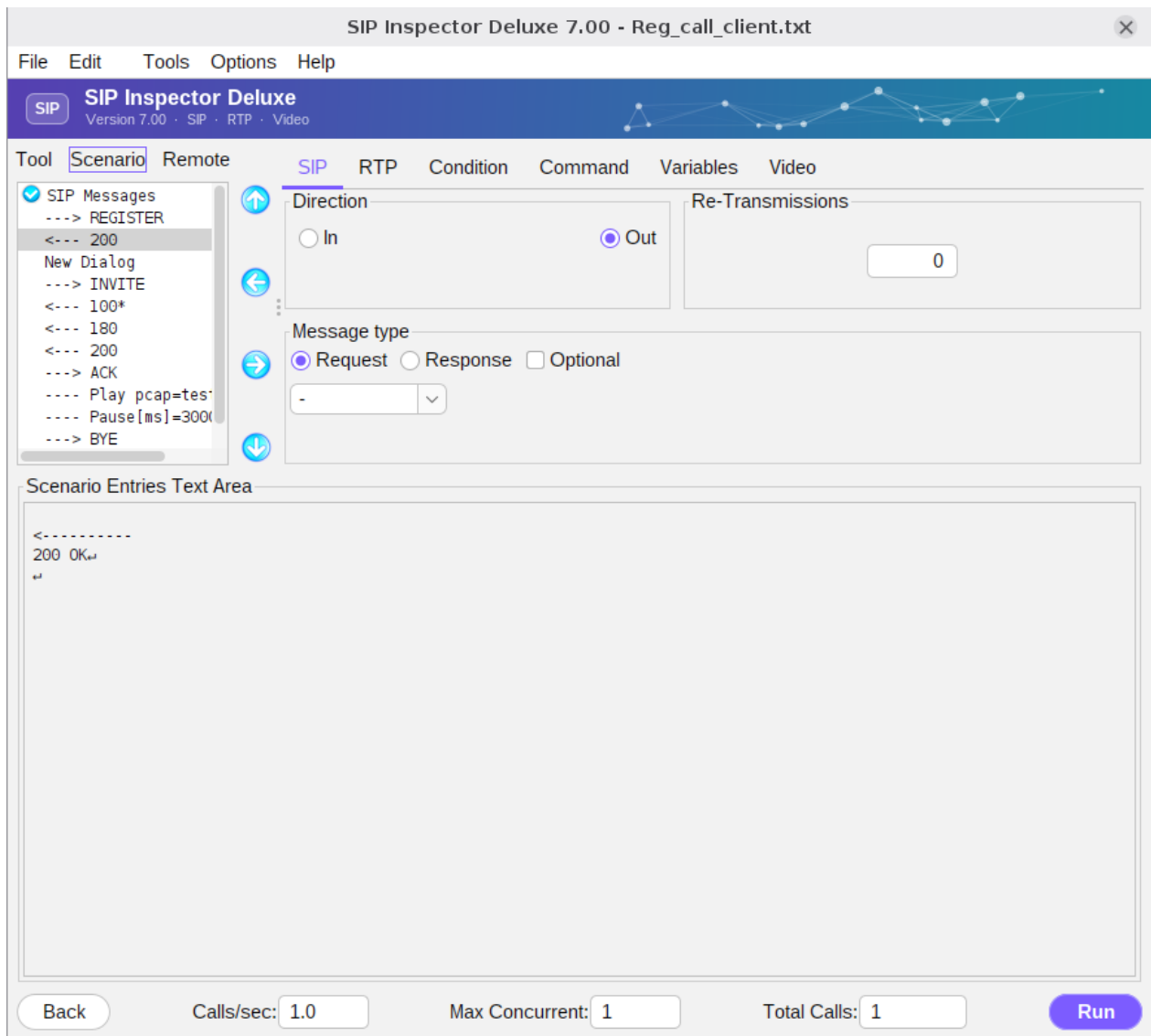


Figure 23 - SIP messages

The direction of a SIP message can be **In** or **Out**. In means that the program receives the message; in the scenario list it has the '<---' symbol. Out means that the message is sent to the remote server or to the remote target and is marked '--->'. Sometimes it is not known in advance how the remote side will respond. That is why optional messages exist: they hint to the tool that a message may or may not come in. Optional messages are marked with an asterisk.

For example, this scenario covers both the case of a server that challenges the REGISTER and the case of a server that simply answers 200 OK:

```
---> REGISTER
<--- 100*           // optional response
<--- 401*           // optional response
---> REGISTER*      // sent only if an optional response was received
<--- 200
```

Msg type is a drop-down list of the standard requests and responses. It does not limit you: to create your own request or response delete the offered text and type the wanted one. **Re-Transmissions** tells the tool how many times to retransmit the message if no response is received (at most 7). Retransmissions follow the RFC timers: 0.5, 1, 2, 4, ... seconds.

Every SIP message of a scenario is a block of plain text, as it goes on the wire, with keywords where a value changes from call to call:

```

----->
INVITE sip:[toUsername]@[remote_ip]:[remote_port] SIP/2.0
Via: SIP/2.0/[transport] [local_ip]:[local_port];branch=[branch]
From: <sip:[fromUsername]@[local_ip]:[local_port]>;tag=[call_number]
To: <sip:[toUsername]@[remote_ip]:[remote_port]>
Call-ID: [call_number]@[local_ip]
CSeq: [cseq+1] INVITE
Max-Forwards: 70
Content-Type: application/sdp
Content-Length: [len]

v=0
...

```

RTP

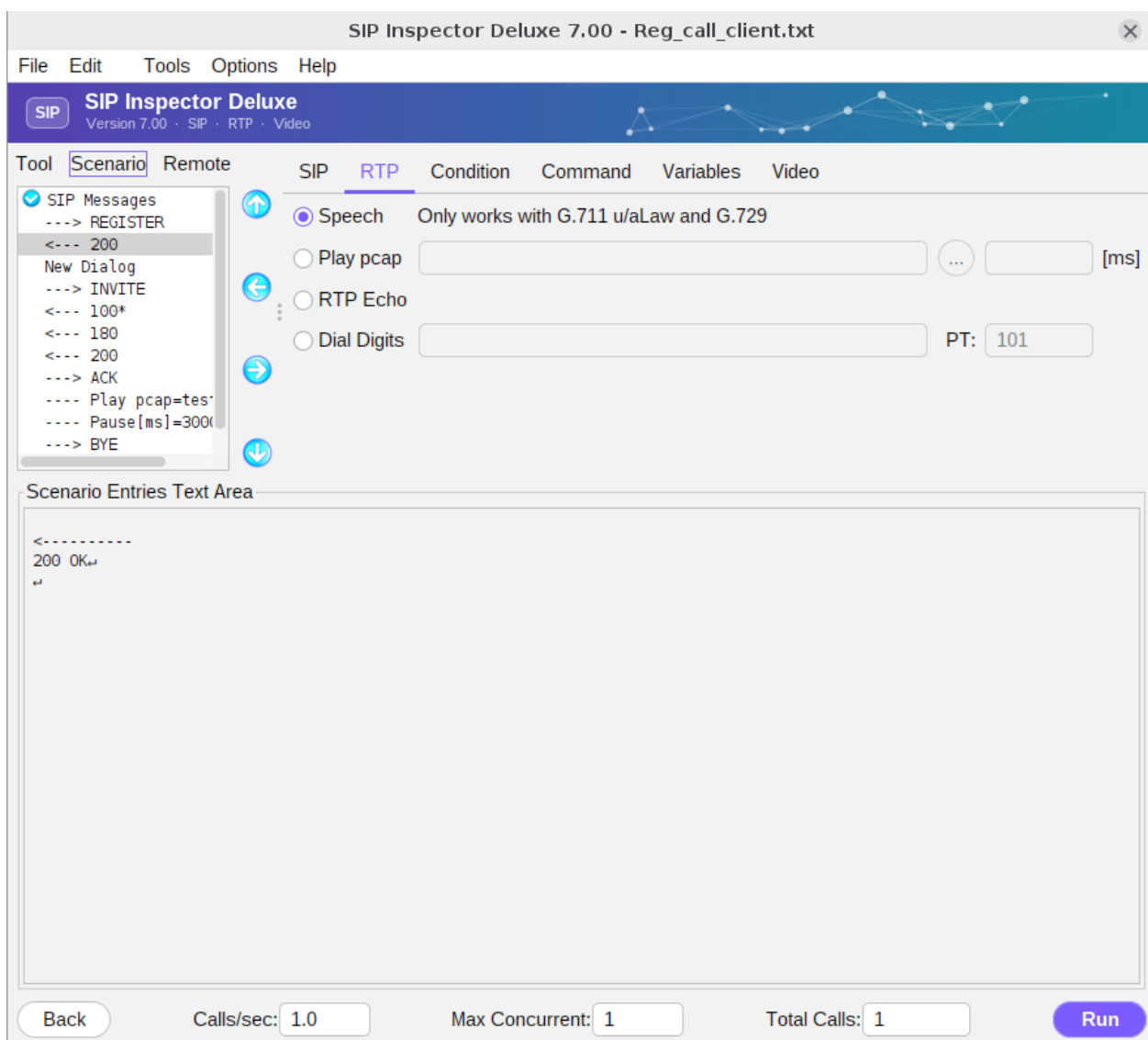


Figure 24 - RTP commands

RTP can be played from a saved pcap file, can be live two-way speech, or incoming RTP can be echoed back to the sender. There is also an option to send out-of-band DTMF digits. Only a scenario with live speech can insert digits. **Speech** works with G.711 u-law and a-law and with G.729. **Play pcap** takes the file and, in the field after it, how long to play it in ms. **RTP Echo** sends back what arrives. **Dial Digits** sends the digits with a payload type (PT). The PT for telephone events is obtained from the incoming SDP. The PT field starts with 101; a value in it forces that payload type, and if the field is left empty digits are sent with payload type 99.

A scenario that plays RTP must keep the call open while the RTP plays. The first scenario below does not produce any audio: as soon as RTP is started BYE is sent, and the end of the call stops the RTP. The second scenario is the proper way: a pause right after 'Play pcap' gives the duration. The third echoes the RTP of the other side:

```
---> INVITE
<--- 100*
<--- 180
<--- 200
---> ACK
---- Play pcap=test.pcap
---> BYE
<--- 200
```

```
---> INVITE
<--- 100*
<--- 180
<--- 200
---> ACK
---- Play pcap=test.pcap
---- Pause[ms]=15000
---> BYE
<--- 200
```

```
<--- INVITE
---> 100
---> 180
---> 200
<--- ACK
---- RTP Echo
<--- BYE
---> 200
```

Sometimes a captured pcap file is shorter than the time during which it has to play. That is not a problem: the next example plays a pcap file for 60 seconds and starts over every time the end of the file is reached.

```
---- Play pcap=test.pcap
---- Pause[ms]=60000
```

If the file has 45 seconds of audio and only the first 15 seconds must play, use a duration. The audio plays for 15 seconds and nothing is sent in the next 45; when the pause expires the scenario goes on:

```
---- Play pcap=test.pcap;duration=15000
---- Pause[ms]=60000
```

Several pcap files can be played at the same time, for example three different codecs sent in parallel for 15 seconds. A series of short files plays digits in the middle of speech:

```
---> ACK
---- Play pcap=test.pcap
---- Play pcap=711aLaw.pcap
---- Play pcap=G729.pcap
---- Pause[ms]=15000
---> BYE
```

```
---> ACK
---- Play pcap=test.pcap;duration=5000
---- Pause[ms]=5000
---- Play pcap=dtmf_1.pcap;duration=200
---- Pause[ms]=200
---- Play pcap=dtmf_2.pcap;duration=200
---- Pause[ms]=200
---- Play pcap=test.pcap
---- Pause[ms]=10000
---> BYE
```

SRTP

SRTP, secure RTP, is seamlessly integrated. Define a scenario like for any other call and indicate in the SDP that SRTP is used: the media line uses **RTP/SAVP** and has a crypto line with the cipher suite and the keying material. The supported suites are AES_CM_128_HMAC_SHA1_80 and AES_CM_128_HMAC_SHA1_32. For an incoming call:

```
<--- INVITE
---> 100
---> 180
---> 200
<--- ACK
---- Play pcap=test.pcap
<--- BYE
---> 200
```

The SDP of the 200 OK must look similar to this:

```
v=0
o=111 843670094 843670094 IN [IP_ver] [local_ip_sdp]
s=-
c=IN [IP_ver] [local_ip_sdp]
t=0 0
a=sendrecv
m=audio [media_port] RTP/SAVP 0 101
a=crypto:1 AES_CM_128_HMAC_SHA1_80 inline:LXr6FMnrukKNw8ZqISVKBk9y8g3/wc37L6le/gWQ
a=rtpmap:0 PCMU/8000
a=rtpmap:101 telephone-event/8000
a=fmtp:101 0-15
a=ptime:20
```

When such a call is established SIP Inspector uses the key that the remote party advertised to decrypt its RTP, and uses its own key to encrypt the RTP it sends, whether the content comes from a pcap file or from live speech. For an outgoing call the logic is the same, but the SRTP information goes into the SDP of the INVITE. The scenarios `SRTP_reg_incoming_call_client.txt` and `SRTP_reg_outgoing_call_client.txt` are complete examples.

Condition and branching

Conditions and branching are important to control the execution of a scenario. In some cases the course of the execution differs depending on which message is received, on the value of a variable or on the number of calls generated so far.

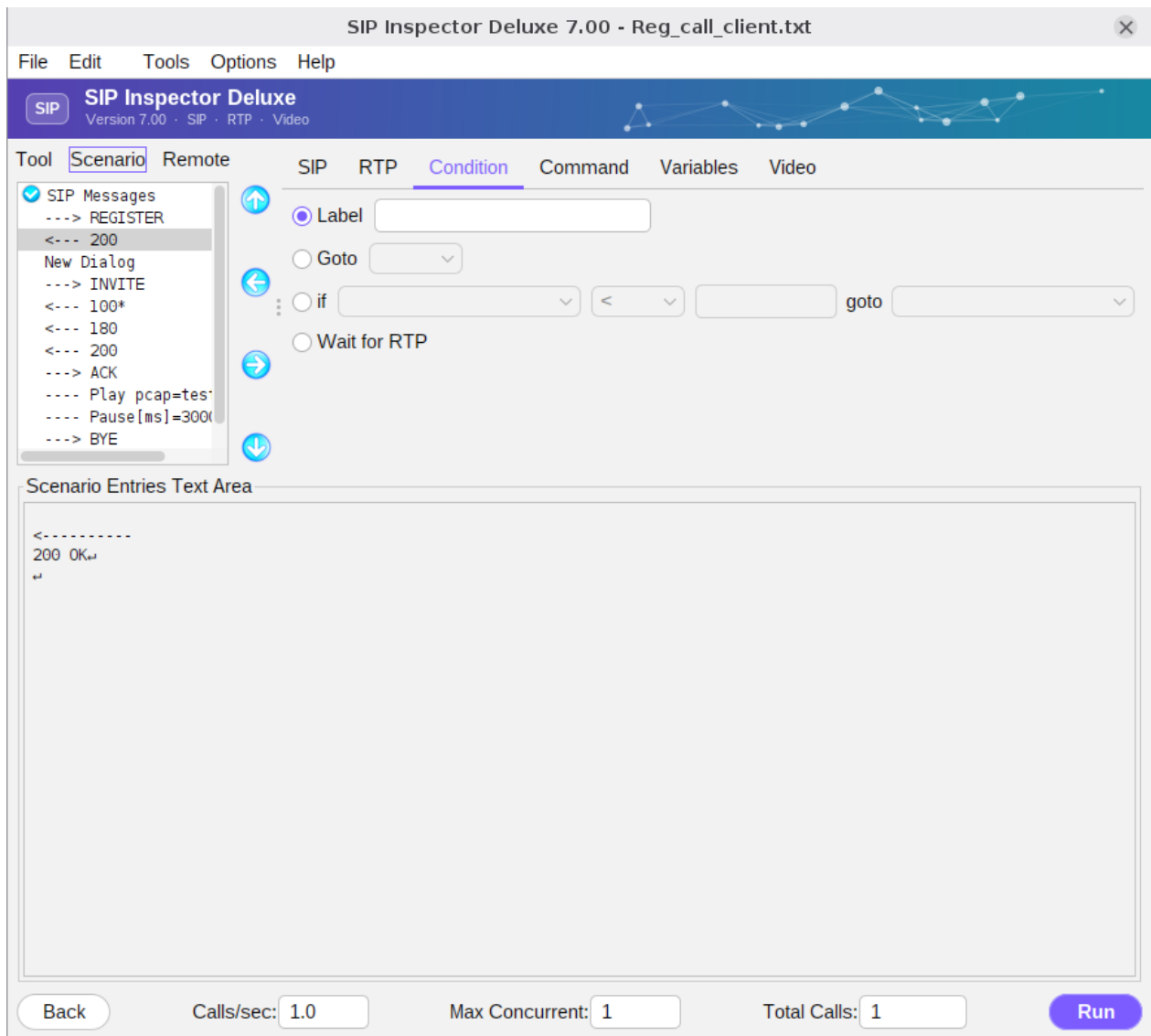


Figure 25 - Condition tab

There are 3 subcategories: **label**, **goto** and **if**. A **label** is a marker in the scenario; its name can be any text, and it is used together with 'goto' and 'if'. When **goto** is selected the tool lists the available labels; pick one and create the entry (the label must already exist). With **if** the tool enables the fields of a condition: the first is a drop-down list of the variables, the second the relational operator, the third the value to compare with, and the fourth the label to jump to when the condition is true. There are two predefined choices, **REQ** and **Resp**, for decisions that depend on the message that came in, for example whether it is a CANCEL request or a 488 response. **Wait for RTP** is the same as the command of the Command tab.

The example processes incoming calls of all users except user 1000. The variable 'user' gets the user name from the From header with a star pattern. If the call comes from 1000 the tool does not answer it, and the scenario goes to its end:

```
<--- INVITE
---- set user=("From","<sip:*@")
---- if (user == 1000) goto end
--> 100
--> 180
--> 200
<--- ACK
---- Pause[ms]=15000
<--- BYE
--> 200
end:
```

The course can also depend on the response of the server. Here a client registers: if it gets 404 the scenario ends, 407 makes it send a REGISTER with digest authentication, and 200 completes the registration:

```
---> REGISTER
---- if (Resp == 404) goto end
<--- 100*
<--- 407*
---> REGISTER*
<--- 200
end:
```

Look at the scenarios in the 'scenarios' folder for more examples, for instance `Reg_branching_client.txt`.

Command

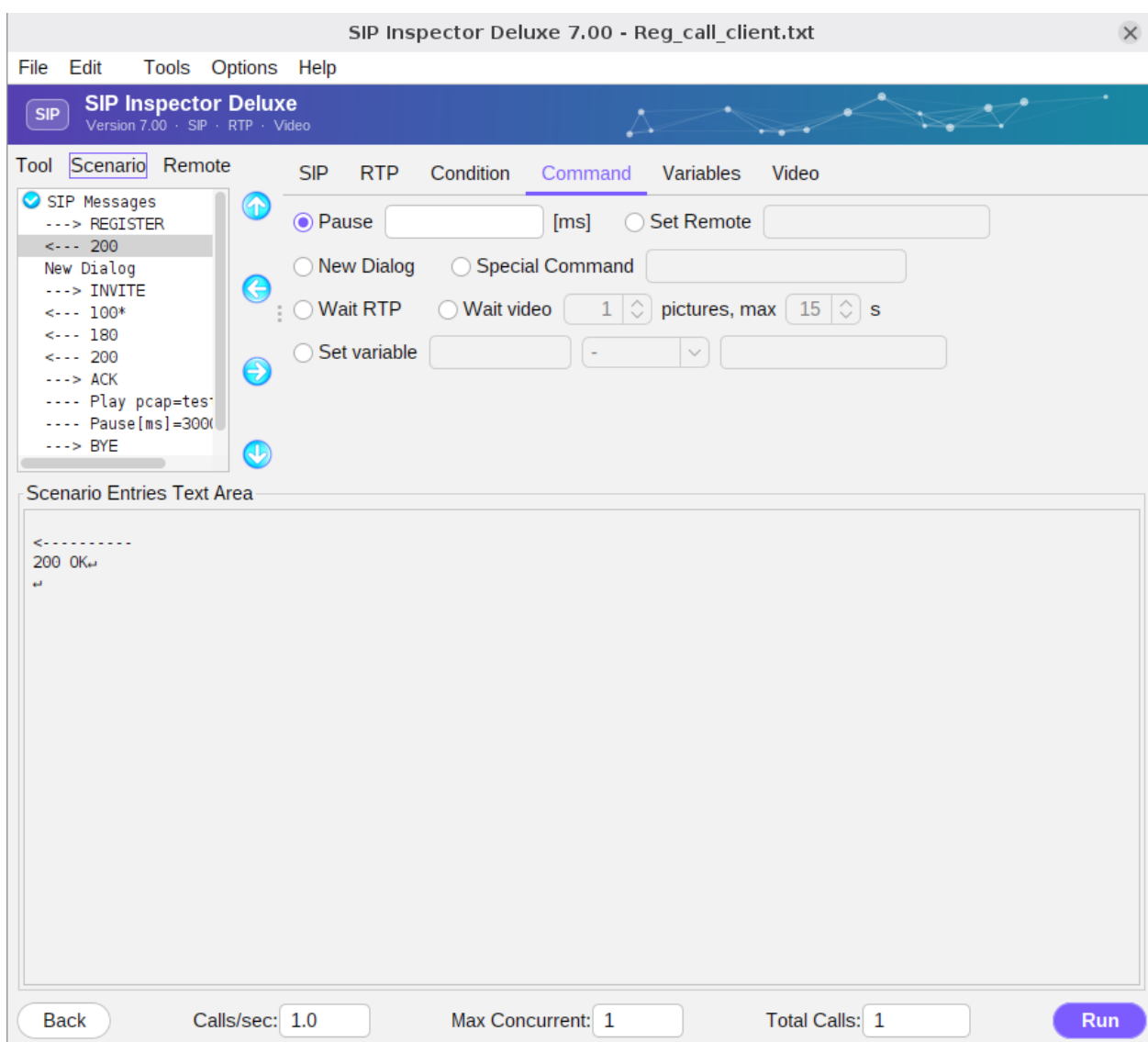


Figure 26 - Command tab

Pause - needs a number of milliseconds, or a keyword such as `[$delay]` that refers to a variable. It puts a pause between scenario entries. To simulate real traffic it is useful to create a variable pause: reference a variable that has a random value.

Set Remote - used when the program has to deal with strict and loose routing, or when it does call brokering between two SIP endpoints. The command sets the destination of the next SIP messages. It can be used many times in one scenario. It is needed for third party call control (3PCC). The remote target can be combined with

values from a file, or can be taken from incoming messages: for loose or strict routing an address is extracted from a Record-Route header, stored in a variable and used as the target. Once a remote target is set, the keyword [remote_ip] is replaced with its address, overriding the setting of the initial screen. The value may include a port, for example 192.168.1.103:5077; without a port 5060 is used.

New Dialog - separates dialogs within the same scenario file. See Multiple dialogs in a scenario.

Special Command - a quick way to add a new command. At the moment there is one: setCC(n) sets the maximum number of concurrent calls while the scenario runs, which is how Load_gradual_increase_client.txt raises the load step by step.

Wait RTP - the scenario goes on when the first RTP packet of the other side arrives. There is no time limit. It waits for audio: video packets do not release it.

Wait video - the scenario goes on when the video of the call has been decoded into the given number of pictures. If that does not happen within the time (15 seconds by default) the call fails with 'No video received'. The time starts when this entry starts, so after a Wait RTP it counts from the first audio packet. If video is not decoded (the Video tab's 'Process video packets' is not ticked, or the CLI runs without -pv) the entry does not wait. See Video.

Set variable - gives a variable a value, which the scenario uses as [\$name]. The header - with an expression that starts with = sets a number or a sum, for example =500 or =[\$a]+1000 (the operators are + - * /). A header such as To:, From:, Call-ID:, Contact: or startline with an expression that has a * takes that part of the header of the last message, for example tag=*. In a scenario file: set_Variable=toTag "To:" "tag=*".

Variables

A variable stores a value. The value can be taken from a SIP message or be set by the user. For example, in a fully established dialog the client must remember the To tag. There is a set of predefined variables ([local_ip], [local_port], [peer_tag_param], ..., also known as keywords) and the user defined variables. System variables are set automatically from the configuration and are used without a '\$' sign. User variables are set explicitly and are used with '\$'. They are not global: they belong to the dialog they are defined in, so each call has its own value of the same variable. Variables are created in the **Variables** tab.

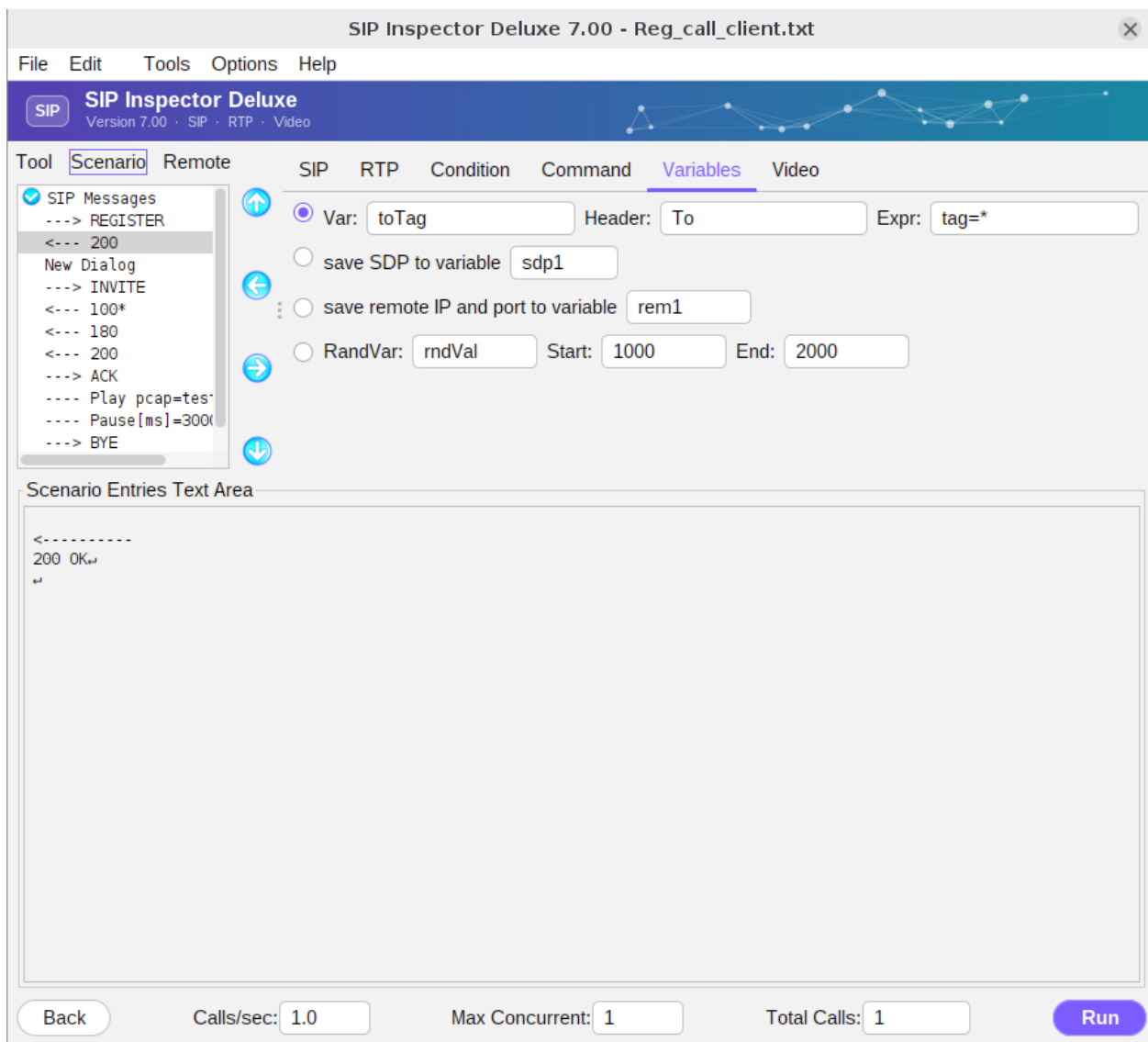


Figure 27 - Variables tab

Field	Meaning
Var	The name of the variable, for example 'toTag'. Later in the scenario the value is used as [\$toTag]. The '\$' tells SIP Inspector that this is a user variable and not a keyword.
Header	The SIP header to take the data from. If left empty the search starts at the beginning of the message and stops at the first match. 'startline' stands for the first line of a message, which has no header name.
Expr	A 'star-matching' expression. Whatever the * matches is stored in the variable. It is much simpler than a regular expression.
save SDP to variable	Stores the whole SDP body in a variable.
save remote IP and port to variable	Stores the IP address and port of the remote end in the form IP:port.
RandVar	Gives the variable a random value between Start and End (both included).

The logic of the star pattern: (a) find the text before the *, find the text after it, and return what is between; (b) if there is nothing before the *, return what comes before the text after it; (c) if there is nothing after the *, return what comes after the text before it. Suppose these pieces must be taken from the header

```
Contact: Zarko Coklin user <sip:112@192.168.1.201:5060;transport=udp;rinstance=124_test_32>
```

Var	Header	Expression	Result	Comment
var0	Contact	"*"	Zarko Coklin user <sip:112@192.168.1.201:5060;transport=udp;rinstance=124_test_32>	The whole header value.
var1	Contact	"Zarko Coklin user *"	<sip:112@192.168.1.201:5060;transport=udp;rinstance=124_test_32>	A * at the end gets the rest.
var2	"Contact"	"* Coklin"	Zarko	A * at the beginning gets the first part. The header name is in quotes.
var3	"Contact:"	" * "	Coklin	The value starts with "Zarko" and not with " Zarko", so the first match of " * " is the one around "Coklin".
var4	Contact:	"<sip:*@"	112	A convenient way to get a user name.
var5	-	"=500"	500	No header: a number is assigned. In the Variables tab only integers can be assigned; the Set variable command also does sums.

A particular instance of a header can be addressed, which is useful for Record-Route: Record-Route-1:, Record-Route-2: and so on. See `Record_route_client.txt`.

Video

Version 7 can offer, send and receive H.264 video together with audio in one call.

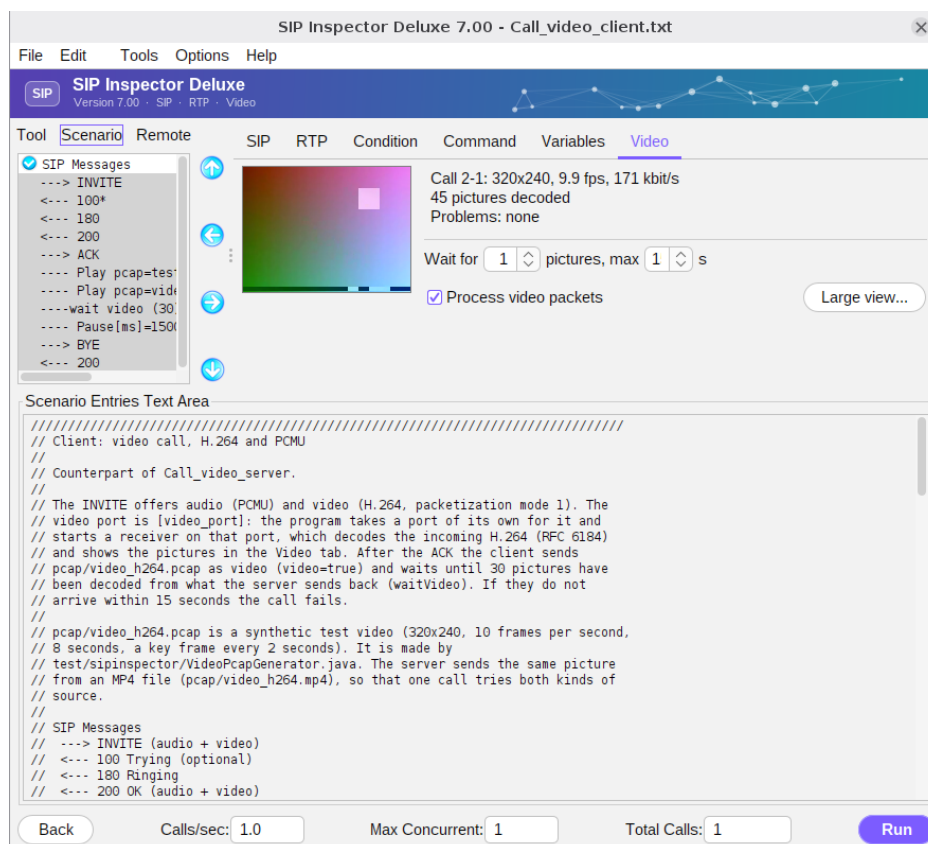


Figure 28 - The Video tab during a video call: the picture received from the other side, with its size, frame rate and bit rate

The Video tab

- The picture area and three lines of text show the received stream: size, frames per second, kbit/s, how many pictures were decoded and the problems found (lost packets and the like).
- **Process video packets.** Decoding takes processing power, so it is off by default: the received video is counted and dropped. The calls still offer a video port, send video and ask the remote side to send it. Tick the box to see the picture. It is the same as the command line option `-pv`.
- **Wait for ... pictures, max ... s** with the left arrow adds a **Wait video** entry to the scenario (the same entry as in the Command tab).
- **Large view...** opens the picture in a window of its own. The scenario screen has a **Video** button for the same (it appears when video is processed).

Video in a scenario

Three things make a video call:

1. **The SDP** has a video media line. The keyword **[video_port]** is the port the program has opened for the video of this call (example below).
2. **Sending video.** A 'Play pcap' entry with `;video=true` sends a pcap file with H.264 RTP as video. The file can also be an MP4 file with H.264 video: the program reads it and sends the H.264 as RTP packets. Nothing is decoded or encoded for that. The audio of an MP4 file is ignored.
3. **Receiving video.** The program starts a receiver on the video port. It puts the packets in order, builds the H.264 pictures (RFC 6184) and decodes them. **Wait video** holds the call until a number of pictures has been decoded.

The media lines of an SDP with audio and video:

```
m=audio [media_port] RTP/AVP 0 101
a=rtpmap:0 PCMU/8000
...
m=video [video_port] RTP/AVP 96
a=rtpmap:96 H264/90000
a=fmtp:96 profile-level-id=420028;packetization-mode=1
a=sendrecv
```

After the ACK a client plays audio and video and waits for the incoming pictures (this is `Call_video_client.txt`; the server `Call_video_server.txt` sends its video from an MP4 file):

```
---> ACK
---- Play pcap=./pcap/test.pcap
---- Play pcap=./pcap/video_h264.pcap;video=true
---- waitVideo=30
---- Pause[ms]=1500
---> BYE
```

Scenario screen - Call_video_client.txt				
Entry	Exp	Unexp	Unexp msgs	
---> INVITE	1	0		
<--- 100*	1	0		
<--- 180	1	0		
<--- 200	1	0		
---> ACK	1	0		
---- Play pcap=test.pcap	1	0		
---- Play pcap=video_h264.pcap;video=true	1	0		
---- wait video (30)	1	0		
---- Pause[ms]=1500	1	0		
---> BYE	1	0		
<--- 200	1	0		

Stop
Start
calls/sec: 0.94
Hide
Video
Close

Figure 29 - The scenario screen of a video call. The Video button shows the picture in its own window

Your own video. Put an MP4 (or MOV) file with H.264 video in the pcap folder and run `test/mp4_info.sh pcap/clip.mp4`. It tells what the program will send (size, frames per second, length, key frames) and prints the video lines for the SDP of the scenario, with the profile-level-id and the sprop-parameter-sets of that file. Files with another codec, without a key frame, cut off or larger than 200 MB are refused.

Note: The video is a test tool for signalling and media paths. A real stream (High profile, 720p and larger, many reference frames) may decode slower or not at all with the decoder that is built into the program. The sent video is never re-encoded, so it always looks like the file.

TLS

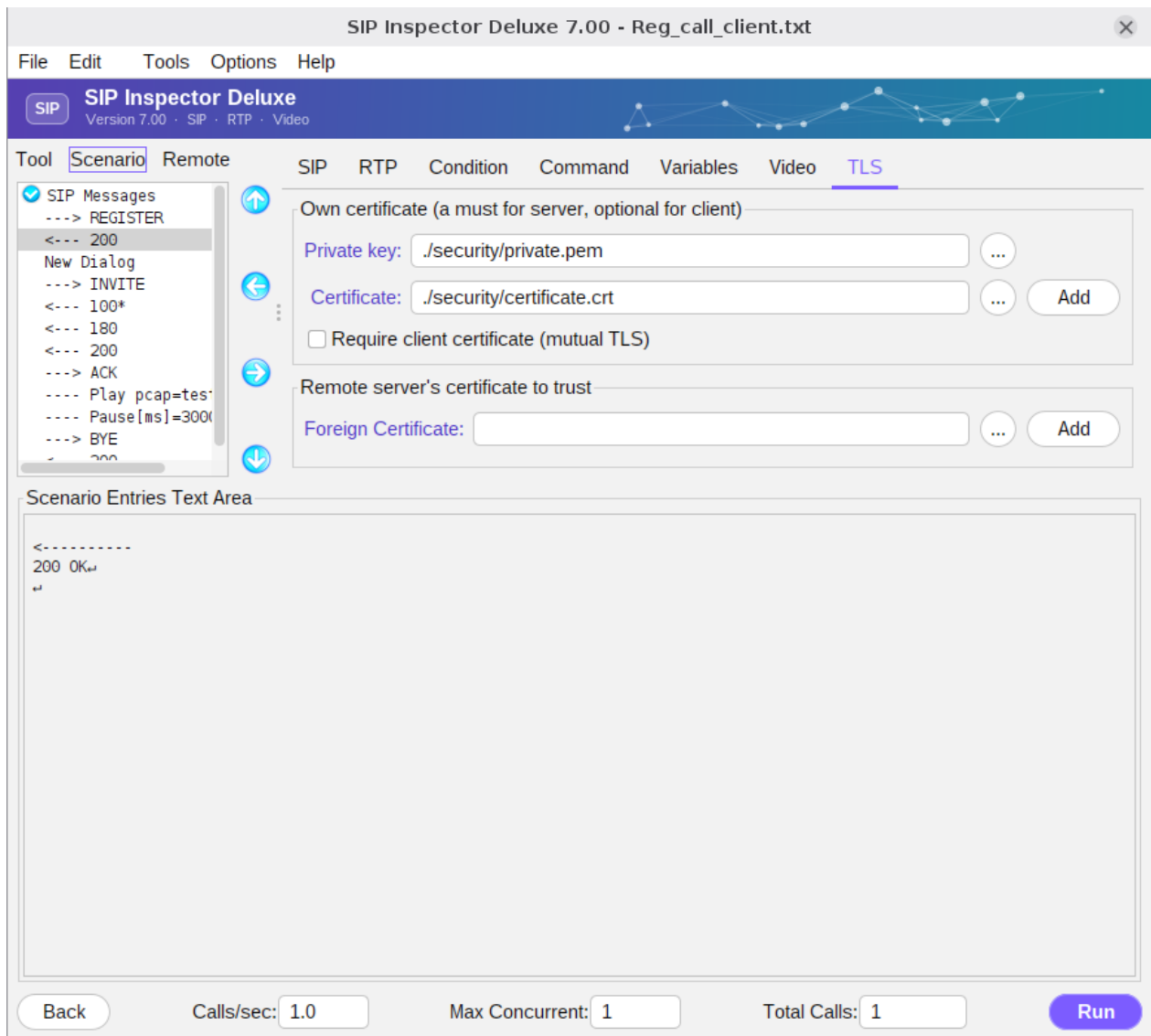


Figure 30 - TLS tab

The TLS tab is presented only when the transport is WSS (secure WebSocket) or TLS. Both need public/private keys and a certificate. All security material is kept in the `security` folder: the keys, the certificate, a keystore (`SIPInspKeystore.jks`) and a truststore (`SIPInspTruststore.jks`). The keystore holds the private key and the certificate of the program; the truststore holds the certificates of remote servers that you choose to trust. If remote certificates are signed by a CA it is not required to add them to the truststore. Both stores can hold several certificates.

If the program runs as a server, its private key and certificate must be added to the keystore. When the keys and the certificate are generated by SIP Inspector this is done automatically; otherwise use the **Add** button of the tab (or `-addCrt`). The certificate has to be added only when it changes. **Check client certificate** forces a server to verify the certificate of the client.

As a client the program trusts self-signed certificates by default, so the remote server's certificate does not have to be added. If something is wrong, delete `SIPInspKeystore.jks` and `SIPInspTruststore.jks` from the security folder: that removes everything the program knows about certificates.

Scenario progress screen

Entry	Exp	Unexp	Unexp msgs
---> REGISTER	8	0	
<--- 200	8	0	
New Dialog	8	0	
---> INVITE	8	0	
<--- 100*	8	0	
<--- 180	8	0	
<--- 200	8	0	
---> ACK	8	0	
---- Play pcap=test.pcap	8	0	
---- Pause[ms]=3000	8	0	
---> BYE	4	0	
<--- 200	4	0	

calls/sec: 1.45

Figure 31 - The scenario progress screen while a client makes calls: the counters of the last entries are lower because those calls are still in their pause

The progress screen shows how well the scenario progresses. Outgoing messages are prefixed with '--->' and incoming messages with '<---'. The columns are:

- **Exp** - how many times the entry was executed as expected: an outgoing message was sent, an incoming message came and matched;
- **Unexp** - how many unexpected messages arrived while the scenario was at this entry. They do not interrupt the execution. Their summary is in **Unexp msgs**, for example '83x200, 11xNOTIFY, 527x401'; hover over the cell to see the whole text.

At any moment the execution can be stopped (**Stop**), started from the beginning (**Start**) or the screen can be closed. **Closing the screen interrupts the scenario execution.** **Hide** puts the screen away by minimizing the main window; restoring the main window brings the progress screen back. If the tool works as a client, once all calls are generated the screen shows the measured speed of call generation (calls/sec). When video is processed there is also a **Video** button. **Call Flow** shows the messages of a call as a ladder diagram, and **Checks** what the SIP/SDP checks found; it turns red with the number of problems. See Call flow and SIP and SDP checks.

Scenario screen - Reg_call_client.txt				
Entry	Exp	Unexp	Unexp msgs	
---> REGISTER	8	0		
<--- 200	8	0		
New Dialog	8	0		
---> INVITE	8	0		
<--- 100*	8	0		
<--- 180	8	0		
<--- 200	8	0		
---> ACK	8	0		
---- Play pcap=test.pcap	8	0		
---- Pause[ms]=3000	8	0		
---> BYE	8	0		
<--- 200	8	0		

Stop
Start
calls/sec: 1.45
Hide
Report...
Call Flow
Checks
Close

Figure 32 - All eight calls are done: every entry shows Exp 8 and Unexp 0

Sometimes calls do not progress as expected and we end up with more unexpected than expected messages. To see what is happening 'on the wire' turn on SIP Logging (Options/SIP Logging). See Logging and troubleshooting.

If several scenario files are loaded (batch processing), the progress screen runs them one after another and the summary of each is saved in a file.

Menu options

There are a few menus. The shortcuts are shown in the menus.

File

File	Edit	Tools	Options
Load Scenario	Ctrl+O		
Load Values	Ctrl+W		
Save Scenario As			
Save Scenario	Ctrl+S		
Exit			

Figure 33 - File menu

- **Load Scenario** - loads a scenario. Several files can be selected (batch processing).

- **Load Values** - loads a values file: one line of values for every call. See Values files: one scenario, many users.
- **Save Scenario** and **Save Scenario As** - save the scenario. After working on a scenario you probably want to keep it and use it next time. Comment lines (//) are kept.
- **Exit**.

Edit

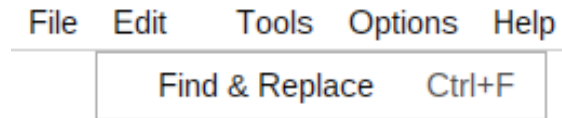


Figure 34 - Edit menu

Find & Replace brings up a dialog where strings are replaced in the loaded scenario. It is a convenient way to change several messages at once, for example the numbers in a default scenario: select all the messages, open Find & Replace and replace the strings.

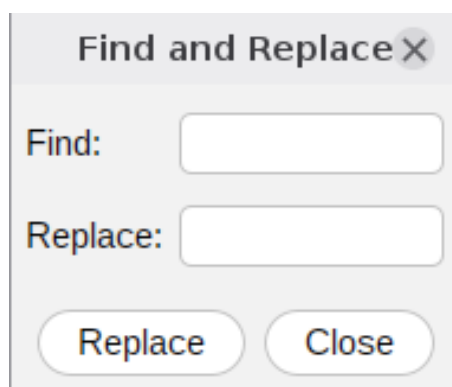


Figure 35 - Find and Replace

Tools

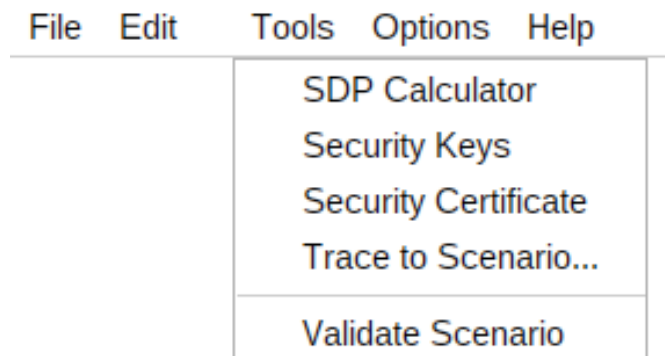


Figure 36 - Tools menu

- **SDP Calculator** - paste an SDP body and it calculates its Content-Length.
- **Security Keys** - creates PKI keys (public and private).
- **Security Certificate** - creates a self-signed certificate.
- **Trace to Scenario** - turns a Wireshark capture into a scenario.
- **Validate Scenario** - checks the SIP messages of the loaded scenario for common mistakes. See Validating a scenario.

Options

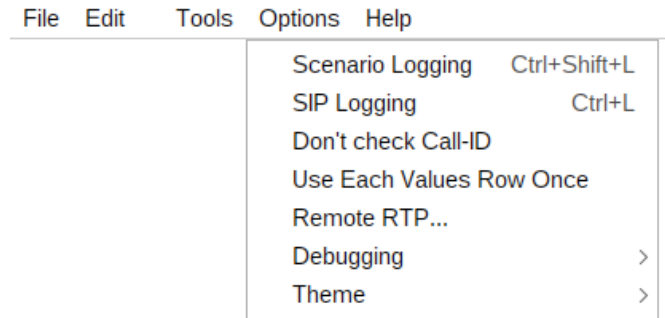


Figure 37 - Options menu

- **Scenario Logging** - saves the progress screen in the Logs folder.
- **SIP Logging** - writes the SIP messages to a log file in the Logs folder.
- **Don't check Call-ID** - used when messages of different dialogs are intertwined. See Don't check Call-ID.
- **Use Each Values Row Once** - every line of the values file is used by one call, then the run ends. See Values files.
- **Remote RTP...** - where the RTP stream goes in an unconditional RTP streaming run.
- **Debugging** - the level of the program log (Error, Warning, Info, Fine or Off). It helps when you report a problem.
- **Theme** - Dark or Light. The choice is kept for the next start.

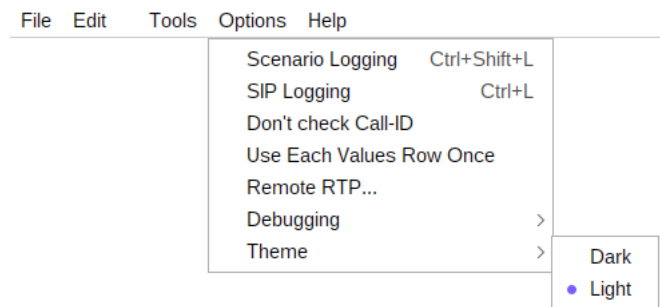


Figure 38 - Options/Theme

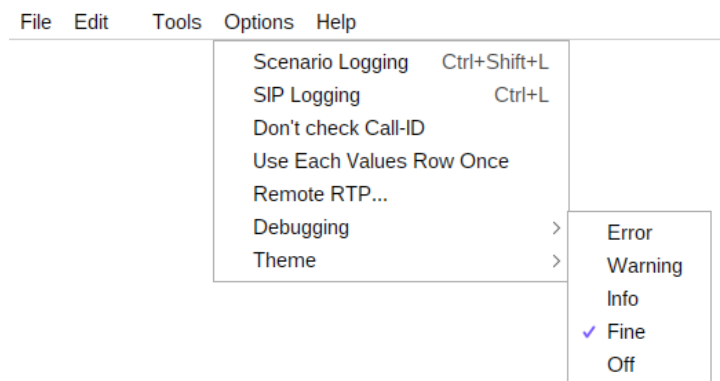


Figure 39 - Options/Debugging

Help

Save Diagnostics - saves the diagnostics bundle: one zip to send with a support question. See The diagnostics bundle.

About - the version, how to contact me, the mailing list and the web site.

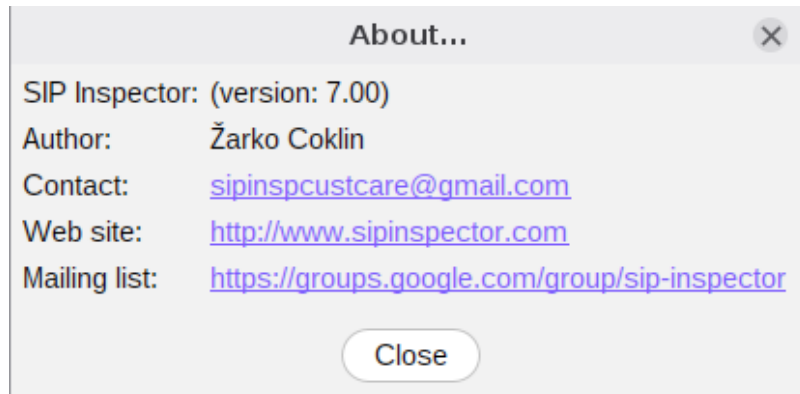


Figure 40 - About

Scenario files, loading and saving

Scenarios can be built in the tabs, as explained above, or they can be loaded from a file. It is wise to save a working scenario in the scenarios folder (File/Save Scenario) and to load it later (File/Load Scenario). The same file runs in the GUI and in the CLI.

A scenario file is plain text. Everything before the first line that contains ----- is a comment, and lines that start with // are comments anywhere. Each entry starts with a line of dashes: -----> is a message the program sends, <----- a message it expects (with a * after it, an optional one), and ----- alone starts any other entry. The entries of the Command, Condition, RTP and Variables tabs look like this in a file:

In the file	Entry
cmd_Pause=2000	Pause for 2000 ms. A keyword such as [\$delay] can be used instead of the number.
cmd_PlayPcap=./pcap/test.pcap;duration=15000;video=true	Play a pcap file (or MP4 file) as RTP; duration and video are optional.
cmd_RTPSpeech	Live speech (RTP Speech).
cmd_RTPEcho	Echo the incoming RTP (RTP Echo).
cmd_DialDigits=1234;PT=101	Send DTMF digits.
cmd_setRemote=192.168.1.110:5060	Send the next messages to this address (Set Remote).
cmd_NewDialog	The messages that follow are a new dialog (New Dialog).
cmd_Spec=setCC(10)	Special command: set the maximum number of concurrent calls.
waitRTP	Wait for the first RTP packet.
waitVideo=30;timeout=15000	Wait for 30 decoded pictures, at most 15000 ms.
set_Variable=toTag "To:" "tag=*"	Variable assignment: name, header, expression.
rndVal=rand[1000000-9999999]	A random value in a range (RandVar).
label=start / goto=start	A label and an unconditional jump.
condition=if (Resp == 404) goto end	A conditional jump.
assert=status 2xx	Pass or fail: the last response is a 2xx. See Assertions and run reports.
assert=rtp loss <= 1	Pass or fail: at most 1% of the RTP that came in was lost.

Scenario comments are kept when a scenario is edited and saved. All scenarios that come with the program start with a comment that says what the scenario does, which scenario is its counterpart and what messages it exchanges. Read them: they are the shortest documentation of the sample scenarios.

Assertions and run reports

A scenario that only sends and waits shows that the other side answered. An **assertion** checks that it answered well: the code of the response, how fast it came, a header, the SDP, the quality of the RTP. It is a scenario entry like any other (Condition tab, **Assert**), placed after the message it checks. When it holds, the scenario goes on. When it does not, the step counts as unexpected with the reason (it shows in the progress table and in the reports), the call is dropped and the CLI ends with exit code 1. An assertion that cannot be read fails too, so a typing mistake cannot make a test pass.

In the file	Passes when
assert=status 200	The last response is a 200. A class (2xx) or several codes (180 183) can be given.
assert=responseTime 500	The last response came at most 500 ms after the last request that was sent.
assert=header Contact exists	The last message has the header (absent: it does not). The short names (m, i, f, t, v ...) count too.
assert=header User-Agent contains Fortinet	A value of the header contains the text (equals: is it; matches: the regular expression is found in it). Case is ignored.
assert=body contains PCMU	The body (SDP) of the last message contains the text; also matches, empty and notEmpty.
assert=rtp packets > 100	The call received more than 100 RTP packets so far.
assert=rtp loss <= 1	At most 1% of the RTP packets were lost (from the RTP sequence numbers).
assert=rtp jitter < 30	The interarrival jitter (RFC 3550) is below 30 ms.
assert=rtp mos >= 3.8	The estimated MOS is at least 3.8.

The RTP numbers are those of the call so far, so put an RTP assertion after a pause that lets the media run, for example just before the BYE. The MOS is an estimate from the loss and the jitter (the simplified E-model of ITU-T G.107 for G.711); it does not know the delay of the network.

scenarios/Call_speech_dials_asserts_client.txt is an example.

Run reports. The numbers of a run are collected for every call: INVITE attempts and answered calls (ASR, the answer-seizure ratio), how long the answered calls lasted (ACD, the average call duration, up to the BYE), the final responses, the time from each request to its final response (min, average, p50, p95, p99, max) and the RTP that came in (packets, loss, jitter, MOS). In the CLI they are printed at the end and written with `- junit`, `- report` and `- html`. In the GUI the **Report...** button of the scenario screen saves them as HTML, CSV or JUnit XML (by the name of the file), and its tooltip shows the numbers of the run so far. The chapter Run reports shows each format with a sample.

Batch scenario processing

SIP Inspector Pro can load several scenario files at the same time (select more than one file in File/Load Scenario). The tool sorts them and runs them alphabetically, one by one; keep that in mind, as the order differs from the order in which you selected the files. As each scenario finishes, the summary of its progress is saved in a file.

Validating a scenario

A scenario written by hand or edited over time can have small mistakes that the other side rejects: a missing header, a From without a tag, a CSeq with the wrong method. **Tools/Validate Scenario** checks the SIP messages of the loaded scenario and reports what it finds. The checks are advisory: the scenario can still be run.

- **Required headers** - every complete SIP message has Via, From, To, Call-ID and CSeq.
- **Tags** - the From header has a tag; a response (except 100 Trying) and a request inside an established dialog have a tag in the To header.

- **CSeq** - a number lower than 2^{31} followed by a method, and in a request the method is the same as the method of the request line.
- **Call-ID** - all the messages of a dialog have the same Call-ID (the dialogs are separated by New Dialog entries).
- **Max-Forwards** - a request that the scenario sends has one.
- **SDP** - the SDP of a message that the scenario sends follows RFC 4566: the lines, their order, c= for every stream, an a=rtpmap for every dynamic payload type, a=crypto only with RTP/SAVP. Keywords get values of their kind for the check ([IP_ver] IP4, an address, a port); an SDP that is a variable ([sdp1]) is checked when the message is sent. See SIP and SDP checks.

Headers that are keywords, for example tag=[call_number] or CSeq: [cseq+1] INVITE, get their value when the message is sent, so they are not checked. Entries that only name a method or a status code ('<--- 200') are not checked either, and the headers that an outgoing response copies from the received request ([last_Via:], [last_To:] ...) count as present.

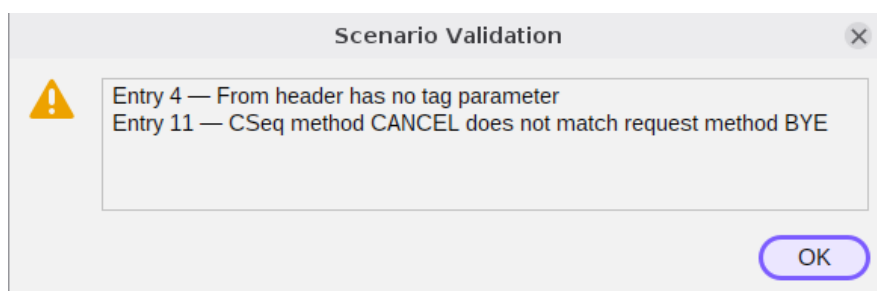


Figure 41 - Tools/Validate Scenario: the report lists the entry numbers and what was found

The entries with findings are marked red in the message tree. Select one and the header line in question is highlighted in the Scenario Entries Text Area. When nothing is found the program says 'No likely SIP compliance issues found.' The marks are cleared as soon as the scenario is changed; run the validation again after fixing it.

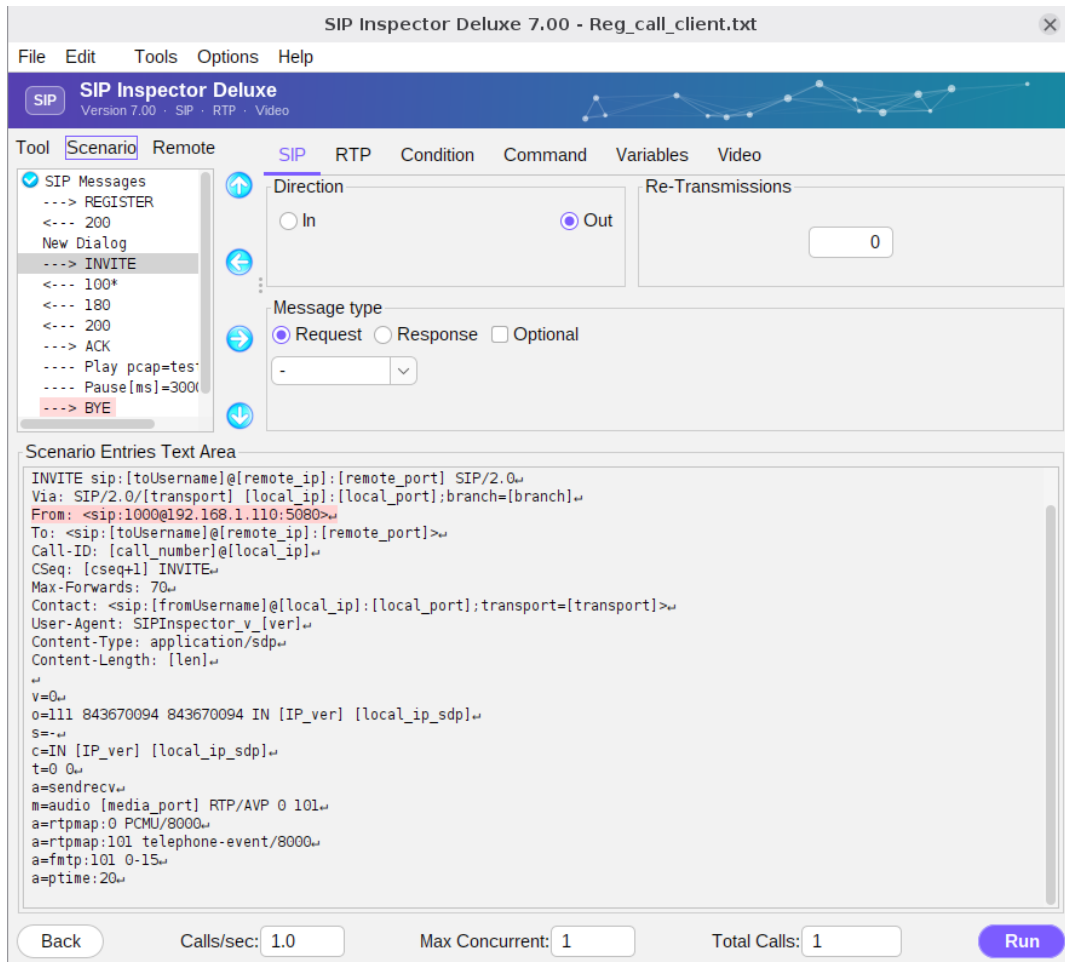


Figure 42 - After the validation: the INVITE (selected) has a From header without a tag, which is highlighted, and the BYE is marked red

Run reports

Every run can be written as a report in four formats. They have the same numbers; each is made for a different reader. The files of the examples below are in the `samples/reports` folder of the program: `passed_run`, 10 calls that all went well, and `failed_run`, 3 calls against a device that answers with a codec that was not offered.

Format	For	How to get it
HTML	People: the result, the numbers, the steps, the SIP/SDP checks and the call flow on one page	GUI Report... (a .html name), Options/Save Run Reports (the Reports folder), -html, GET /report
CSV	Spreadsheets: the totals as # name,value lines, then a table of the steps	GUI Report... (a .csv name), -report, GET /report?format=csv
JUnit XML	CI servers (Jenkins, GitLab, Azure DevOps, GitHub Actions): every step and every gate limit is a test case	GUI Report... (a .xml name), -junit, GET /report?format=junit
JSON	Scripts and tools: the verdict and why, the gate, the numbers, the steps, the checks and the call flow	-json, GET /metrics, GET /report?format=json

```
SIPInspDlx -lip 10.0.0.5 -rip 10.0.0.9 -fs scenarios/Reg_call_client.txt -tc 10 -cps 2 -mc 5 -np \
-gate "asr>=99,p95<=300,mos>=4.0,problems==0" \
-html passed_run.html -report passed_run.csv -junit passed_run.xml -json passed_run.json
```

The HTML report

The page starts with the scenario and the verdict, PASS or FAIL, and why it failed. With a gate, a table shows every limit, the number of the run and whether it held. Then come the calls (started, still active, dropped, INVITE attempts, answered, ASR, ACD), the response times, the RTP that came in (packets, loss, jitter, MOS), the final responses to the INVITEs and every step of the scenario. The page needs nothing but a browser and follows its light or dark theme.

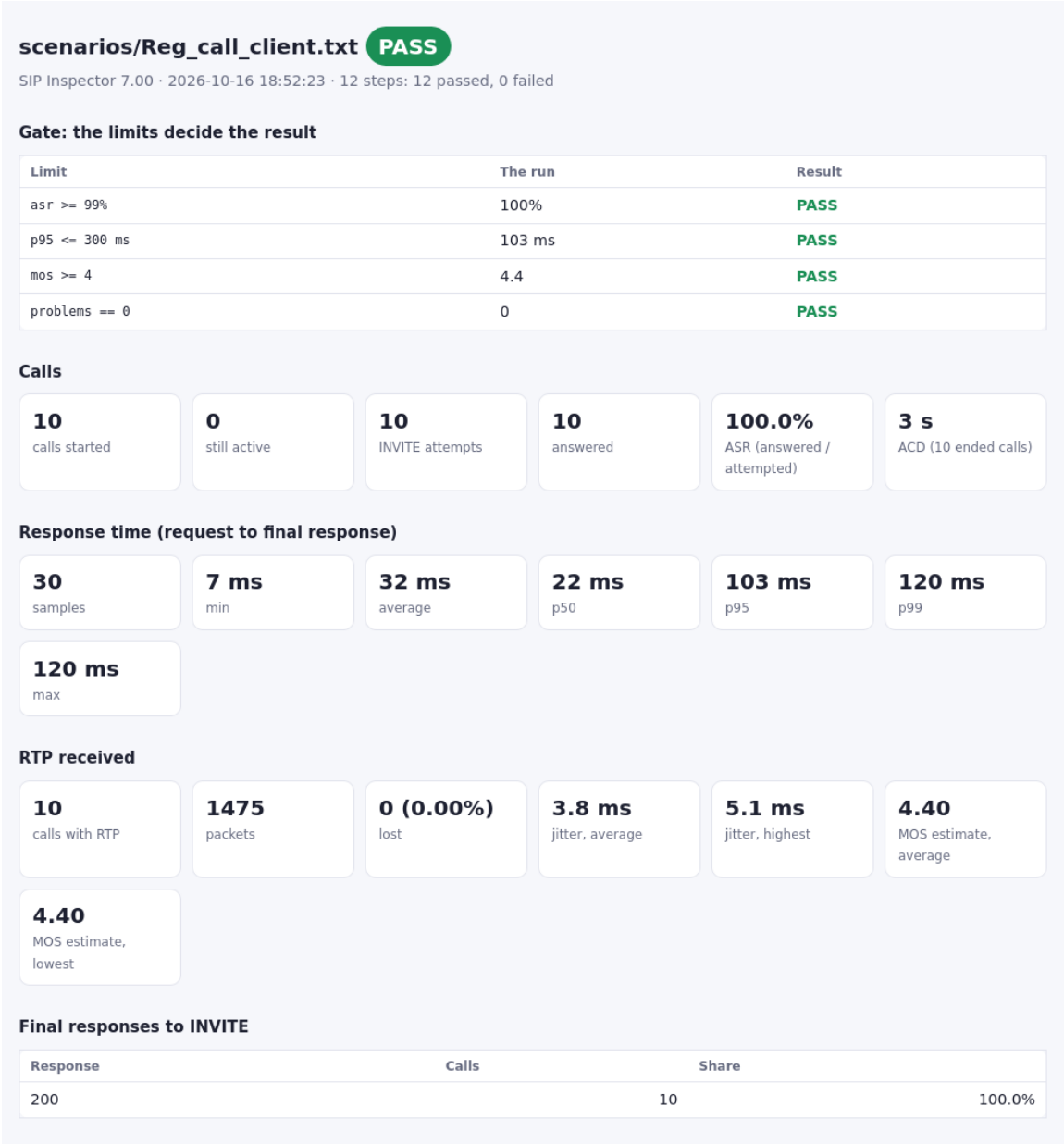


Figure 43 - The HTML report of passed_run: the gate held, 10 calls were answered, the response times and the RTP quality

A run that failed says why at the top. In the steps the failed one is red with the reason, and the steps after it that no call reached are **NOT REACHED**, so it is clear where the calls stopped.

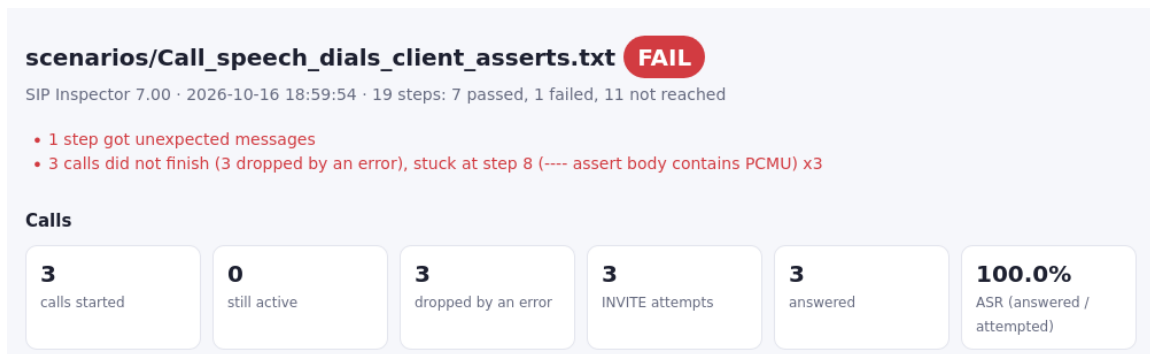


Figure 44 - The top of failed_run: FAIL, and why

Steps					
#	Step	Result	Expected	Unexpected	Unexpected messages
1	---> INVITE	PASS	3	0	
2	<--- 100*	PASS	3	0	
3	<--- 180	PASS	3	0	
4	<--- 200	PASS	3	0	
5	---- assert status 2xx	PASS	3	0	
6	---- assert responseTime 2000ms	PASS	3	0	
7	---- assert header Contact exists	PASS	3	0	
8	---- assert body contains PCMU	FAIL	0	3	3xbody contains PCMU: not found in the body
9	---> ACK	NOT REACHED	0	0	
10	---- RTP Speech	NOT REACHED	0	0	
11	---- Pause[ms]=9000	NOT REACHED	0	0	
12	---- rndVal=rand[1000000-9999999]	NOT REACHED	0	0	
13	---- Dial={\$rndVal}	NOT REACHED	0	0	
14	---- Pause[ms]=11000	NOT REACHED	0	0	
15	---- assert rtp packets > 100	NOT REACHED	0	0	
16	---- assert rtp loss <= 2	NOT REACHED	0	0	
17	---- assert rtp mos >= 3.5	NOT REACHED	0	0	
18	---> BYE	NOT REACHED	0	0	
19	<--- 200	NOT REACHED	0	0	

Figure 45 - The steps of failed_run: the assertion of step 8 failed in all 3 calls, the rest of the scenario was not reached

The page ends with the SIP/SDP checks and the call flow of the first call that did not reach the end. A click on an arrow opens its message: here the 200 OK, whose SDP has G.722 (payload type 9) where the client offered PCMU. That is both why the assertion failed and what the checks found.

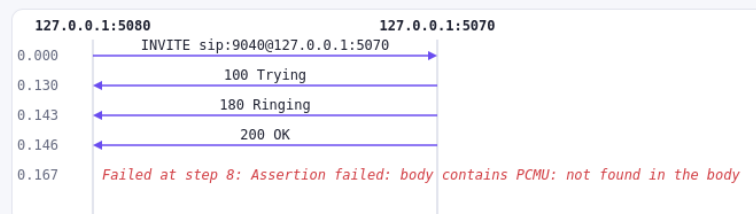
SIP and SDP checks

3 problems (1 kind) in 12 messages. Warnings: they fail the run only with a gate (problems==0).

Direction	Message	Problem	Times	First one
received	200 (INVITE)	offer/answer: m=audio: the answer has g722/8000 (payload type 9), which the offer did not have	3	► Call-ID 2-1

Call flow: the first call that did not reach the end of the scenario

Call-ID 2-1 · **Failed at step 8: Assertion failed: body contains PCMU: not found in the body** · a click on an arrow opens its message



► 1. sent INVITE sip:9040@127.0.0.1:5070 SIP/2.0

► 2. received SIP/2.0 100 Trying

► 3. received SIP/2.0 180 Ringing

▼ 4. received SIP/2.0 200 OK

```

SIP/2.0 200 OK
Via: SIP/2.0/UDP 127.0.0.1:5080;branch=z9hG4bK20
From: <sip:200@127.0.0.1:5080>;tag=2-1
To: <sip:9040@127.0.0.1:5070>;tag=2-1
Call-ID: 2-1@127.0.0.1
CSeq: 2 INVITE
Contact: <sip:127.0.0.1:5070;transport=UDP>
User-Agent: SIPInspector_v_7.00
Content-Type: application/sdp
Content-Length: 210
  
```

```

v=0
o=111 843670094 843670094 IN IP4 127.0.0.1
s=-
c=IN IP4 127.0.0.1
t=0 0
a=sendrecv
m=audio 30000 RTP/AVP 9 101
a=rtpmap:9 G722/8000
a=rtpmap:101 telephone-event/8000
a=fmtp:101 0-15
a=ptime:20
  
```

The MOS is an estimate from the loss and the jitter (simplified E-model for G.711); it does not include the network delay.

Figure 46 - The end of failed_run: the check that found the wrong codec, and the call flow with the 200 OK opened

The CSV report

The first lines are the totals of the run, one # name, value line each, with the same names as the properties of the JUnit report, the JSON metrics and the Prometheus metrics of the API. Then comes a table with one row per step. A spreadsheet opens it as it is; a script reads the totals from the lines that start with #. passed_run.csv, a few of its total lines left out:

```

# scenario,"scenarios/Reg_call_client.txt"
# callsStarted,"10"
# callsActive,"0"
# steps,"12"
# stepsFailed,"0"
# stepsNotReached,"0"
# result,"PASS"
# resultReasons,""
# gate,"asr>=99,p95<=300,mos>=4.0,problems==0"
# gate.asr>=99,"PASS 100%"
# gate.p95<=300,"PASS 103 ms"
# gate.mos>=4,"PASS 4.4"
# gate.problems==0,"PASS 0"
# notCounted,""
...
# calls.attempted,"10"
# calls.answered,"10"
# calls.asrPercent,"100.00"
# calls.acdMs,"3100"
# responseTime.p95Ms,"103"
# rtp.mosAvg,"4.40"
# checks.messages,"90"
# checks.problems,"0"
stepIndex,step,result,expected,unexpected,unexpectedMessages
1,"---> REGISTER",PASS,10,0,""
2,"<--- 200",PASS,10,0,""
3,"New Dialog",PASS,10,0,""
4,"---> INVITE",PASS,10,0,""
5,"<--- 100*",PASS,10,0,""
6,"<--- 180",PASS,10,0,""
7,"<--- 200",PASS,10,0,""
8,"---> ACK",PASS,10,0,""
9,"---- Play pcap=test.pcap",PASS,10,0,""
10,"---- Pause[ms]=3000",PASS,10,0,""
11,"---> BYE",PASS,10,0,""
12,"<--- 200",PASS,10,0,""

```

The JUnit XML report

A CI server shows a JUnit report as tests: the suite is the scenario, and every step is a test case that passed, failed (with the reason) or was skipped (no call got to it). Test case 0 says whether every call reached the end of the scenario. With a gate, every limit is a test case too, and only the limits decide: a step with a few failed calls is then not a failure of the suite. The totals are the properties of the suite. failed_run.xml, without its properties and the classname of every test case:

```

<?xml version="1.0" encoding="UTF-8"?>
<testsuite name="scenarios/Call_speech_dials_asserts_client.txt" tests="20" failures="2" errors="0"
  skipped="11">
  <properties>
    ...
  </properties>
  <testcase name="0 every call reaches the end of the scenario">
    <failure message="3 calls did not finish (3 dropped by an error), stuck at step 8 (---- assert
      body contains PCMU) x3">3 calls did not finish (3 dropped by an error), stuck at step 8
      (---- assert body contains PCMU) x3</failure>
  </testcase>
  <testcase name="1 ---&gt; INVITE">
  </testcase>
  ...
  <testcase name="8 ---- assert body contains PCMU">
    <failure message="expected 0, unexpected 3: 3xbody contains PCMU: not found in the
      body">expected 0, unexpected 3: 3xbody contains PCMU: not found in the body</failure>
  </testcase>
  <testcase name="9 ---&gt; ACK">
    <skipped message="no call got to this step"/>
  </testcase>
  ...
</testsuite>

```

A CI job runs the test and hands the report to the CI server; the exit code of the program fails the job. In Jenkins and GitLab:

```
// Jenkinsfile
sh 'SIPInspDlx -lip 10.0.0.5 -rip 10.0.0.9 -fs scenarios/Reg_call_client.txt -tc 100 -np -junit
    results/sip.xml'
junit 'results/sip.xml'

# .gitlab-ci.yml
sip-test:
  script:
    - SIPInspDlx -lip 10.0.0.5 -rip 10.0.0.9 -fs scenarios/Reg_call_client.txt -tc 100 -np -junit
      sip.xml
  artifacts:
    when: always
    reports:
      junit: sip.xml
```

The JSON report

result is PASS or FAIL and reasons says why; gate has every limit with the number of the run (null without a gate); metrics has the totals; checks what the SIP/SDP checks found; callFlow the first line of every message of the call that the HTML report shows; steps every step. The start of failed_run.json and its checks and call flow:

```
{
  "scenario": "scenarios/Call_speech_dials_asserts_client.txt",
  "result": "FAIL",
  "reasons": ["1 step got unexpected messages", "3 calls did not finish (3 dropped by an error),
    stuck at step 8 (---- assert body contains PCMU) x3"],
  "gate": null,
  "metrics": {
    "callsStarted": 3,
    ...
  },
  "checks": {"messages": 12, "problems": 3, "unlisted": 0, "list": [
    {"direction": "received", "message": "200 (INVITE)", "problem": "offer/answer: m=audio: the
      answer has g722/8000 (payload type 9), which the offer did not have", "count": 3, "callId":
        "2-1"}
  ]},
  "callFlow": {"callId": "2-1", "problem": "Failed at step 8: Assertion failed: body contains PCMU:
    not found in the body", "shown": "the first call that did not reach the end of the
    scenario", "events": [
    {"ms": 0, "kind": "sent", "from": "127.0.0.1:5080", "to": "127.0.0.1:5070", "line": "INVITE
      sip:9040@127.0.0.1:5070 SIP/2.0"},
    {"ms": 130, "kind": "received", "from": "127.0.0.1:5070", "to": "127.0.0.1:5080", "line":
      "SIP/2.0 100 Trying"},
    ...
    {"ms": 167, "kind": "note", "from": "127.0.0.1:5080", "to": "127.0.0.1:5080", "line": "Failed
      at step 8: Assertion failed: body contains PCMU: not found in the body"}
  ]},
  "steps": [
    ...
  ]
}
```

Call flow

Call Flow in the scenario screen shows the messages of a call as a ladder diagram, while the run goes on: one line for each end of the call, one arrow for each message, with the time from the first message. Click a message, or move with the arrow keys, and it is shown whole below. A message sent again because no answer came is a dashed arrow, a response of 400 and up is red, and why a call did not reach the end of the scenario (a time out, a failed step or assertion, the run stopped) is the last line, in red.

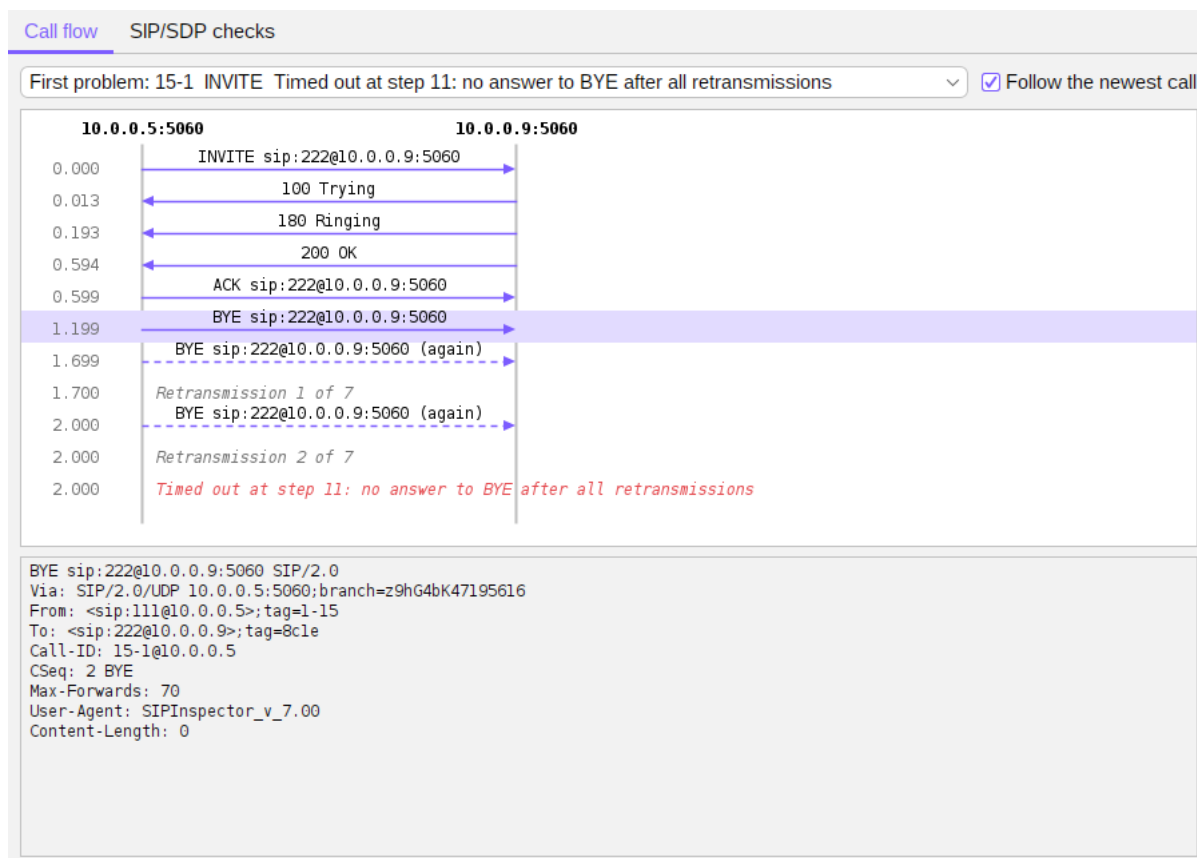


Figure 47 - Call flow of a call whose BYE got no answer: the BYE is selected, it was sent three times

The list above the ladder has the calls that are going on, the last 20 calls that ended, and the **first problem**: the first call of the run that did not reach the end, which is kept for the whole run. **Follow the newest call** shows the call that started last as calls come and go; picking a call in the list turns it off. A call keeps its first 200 lines.

The run reports have the call flow too. The HTML report (and the report of Report..., -html and the API) ends with the ladder of the first call that did not reach the end, else of a call that was still going at the end, else of the last call; a click on an arrow opens its message. The JSON report has it under callFlow. A CI job that fails can be looked at without running it again.

SIP and SDP checks

Every message that a call sends and receives is checked against the SIP and SDP rules: what the other side gets wrong, and what a scenario sends wrong. What is found is a **warning**. It does not change the result of the run: a device that sends a 200 OK without a To tag may still be what you want to test. A gate with problems==0 makes a run pass only with clean messages.

What	The checks
SIP (RFC 3261)	The start line; Via, From, To, Call-ID, CSeq and in a request Max-Forwards; a header that must be there once and is there twice; the branch of the Via starts with z9hG4bK; the tag of From, and of To in a response (except 100); CSeq number and method; Contact in an INVITE and its 2xx; Content-Length (over TCP and TLS it must be there) and Content-Type of a body; CRLF line ends
SDP (RFC 4566)	v=0 first; o=, s= and t= once and in their order; c= for the session or every stream, with an address of its type; m= with a port and payload types of 0 to 127; an a=rtpmap for every dynamic payload type, and none for a payload type that is not offered; a=crypto only with RTP/SAVP, and RTP/SAVP with a key
Offer/answer (RFC 3264)	The answer has as many m= lines as the offer, in the same order; no codec that the offer did not have (compared by name, so another payload type number is fine); a direction that fits the offer (sendonly is answered with recvonly or inactive); an SRTP stream is accepted with a key. A late offer (in the 200) answered in the ACK is checked too

The same problem of the same kind of message is counted, not listed again: a run of 10 000 calls with one wrong header has one line that says 10000. The console prints the first five; the HTML and JSON reports, the diagnostics bundle and the **Checks** button of the scenario screen have all of them, each with the first message that had it. The program log has a line for every new kind.

```
sip/sdp checks: 1 problem (1 kind) in 9 messages
  received 200 (INVITE): offer/answer: m=audio: the answer has g722/8000 (payload type 9), which
    the offer did not have
```

Call flow SIP/SDP checks (100)			
SIP/SDP checks: 100 problems (4 kinds) in 75 messages. They are warnings: the run fails on them only with a gate (problems==0).			
Direction	Message	Problem	Times
received	200 (INVITE)	Content-Length is 115, but the body has 113 bytes (cut short?)	25
received	200 (INVITE)	SDP: the payload type 101 of m=audio has no a=rtpmap	25
received	200 (INVITE)	offer/answer: m=audio: the answer has g722/8000 (payload type 9), w...	25
received	180 (INVITE)	the To header of the response has no tag	25

First one, Call-ID 20-1:

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP 10.0.0.5:5060;branch=z9hG4bK-1695272
From: <sip:111@10.0.0.5>;tag=1-15
To: <sip:222@10.0.0.9>;tag=8c1e
Call-ID: 20-1@10.0.0.5
CSeq: 1 INVITE
Contact: <sip:222@10.0.0.9:5060>
Content-Type: application/sdp
Content-Length: 115

v=0
```

Figure 48 - The checks of a run: a device answers with a codec that was not offered, and gets three more things wrong

The checks cost little, but in a load test where every bit of CPU counts they are turned off with `-noChecks` (`noChecks` in `POST /run`). `Validate Scenario` has the same SDP rules for the messages of a scenario before it is run.

Values files: one scenario, many users

A scenario describes one call. Real tests need many calls that are *not* the same: 500 phones that each register with their own user name and password, calls from many different numbers, a sweep over a range of ports. Writing one scenario for every user does not scale. A **values file** solves this: the scenario is written once with placeholders, `[field0]`, `[field1]` ..., and the values file is a table with one line for every call. Each new call takes the next line and puts its values in place of the placeholders.

In short: the values file is the way to turn one scenario into a load test with real accounts, or into a sweep over numbers, ports or addresses. It needs no programming: a scenario with `[fieldN]` keywords, a text file and `File/Load Values`.

How it works

1. **The scenario** uses `[field0]` where the first value goes, `[field1]` for the second, and so on. The keywords can be anywhere in a message the program sends: the request line, any header, the body, and inside the authentication keyword.
2. **The values file** has one line for every call. The values on a line are separated with `;`: the first is `[field0]`, the second `[field1]` ...
3. **Load it** with `File/Load Values` (or `-fv` on the command line) and press Run. Every new call takes the next line. When the last line is used the program starts again at the first line, unless **Options/Use Each Values Row Once** is on (see below).

```

----->
REGISTER sip:[remote_ip] SIP/2.0
From: <sip:[field0]@[remote_ip]>;tag=...
To: <sip:[field0]@[remote_ip]>
...
Proxy-Authorization: [authentication
username=[field1]; password=[field2];]

```

```

//[field0];[field1];[field2]
107;user107;pass107
108;user108;pass108
109;user109;pass109

```

The scenario (left, from scenarios/Reg_series_client.txt) and its values file (right, values/Reg_series_values.txt)

Call	Line used	[field0]	[field1]	[field2]	The REGISTER is sent for
1	1	107	user107	pass107	107, authenticated as user107 / pass107
2	2	108	user108	pass108	108, authenticated as user108 / pass108
3	3	109	user109	pass109	109, authenticated as user109 / pass109

A call takes its line when it first needs a value and keeps it to the end: the INVITE, the ACK and the BYE of one call always carry the same values, even when many calls run at the same time. A call that is a whole scenario run (with New Dialog entries, several dialogs) uses one line for all its dialogs.

Step by step: register many users

The sample files register a list of users against a registrar or a proxy that asks for authentication.

1. Load the scenario: File/Load Scenario, scenarios/Reg_series_client.txt.
2. Load the values: File/Load Values, values/Reg_series_values.txt. Edit it first so that it has your users and passwords, one user per line.
3. Turn on **Options/Use Each Values Row Once**: every user registers once, and the run ends after the last line. Total Calls can stay 0.
4. Set Calls/sec (for example 10) and Max Concurrent (for example 50), then press **Run**. The progress screen counts the REGISTERS and the 200 OKs; one 200 per line means every user is registered.

The same from the command line:

```

SIPInspDlx -lip 10.0.0.5 -rip 10.0.0.1 -fs scenarios/Reg_series_client.txt \
-fv values/Reg_series_values.txt -fvo -cps 10 -mc 50

```

Writing a values file

Rule	Example
One line for every call; values separated with ;	1000;secret1000;Alice
Lines that start with // or # are comments. Use the first line to name the columns.	//[field0];[field1];[field2]
A value can be empty	1000;;Alice ([field1] is empty)
A ; inside a value is written \;	sip:1000@host\;transport=tcp
Spaces belong to the value, so do not put spaces around ;	1000;Alice Smith
The file is read as UTF-8 (letters like Đ, ö, é are fine)	2001;Đorđe
A value that counts: start+step (see Counting values)	5000000+1
Name the columns with a line #columns:, then [field:user] works as well as [field0]	#columns: user;password;name

Column names. With named columns a scenario says what it means: [field:user] instead of [field0]. The numbers keep working, and names ignore case. A name that is not a column gives an empty value; when the values file is loaded and before every run the program checks the names of the scenario against the columns, and the CLI prints a warning such as *the scenario uses [field:password], but the values file has no column password (its columns: number, user, password)*.

CSV files. A values file whose name ends with .csv is read the way spreadsheets save it: values separated with commas, the **first line names the columns**, and a value in double quotes may have commas, line breaks and "" for a quote. Lines that start with # are comments. A .csv file without a first line of names can have a #columns: line instead. In an XML values file the root element names the columns: <values columns="user;password">.

```
number,user,password,display name
2001,alice,"se,cret","Smith, Alice"
2002,bob,pw2,Bob

From: "[field:display name]" <sip:[field:number]@[remote_ip]>;tag=[call_number]
Proxy-Authorization: [authentication username=[field:user]; password=[field:password];]
```

Values that span several lines or contain XML are easier to write in an XML values file. Each <rowValues> is a line, each <value> a column; CDATA keeps the text exactly as written:

```
<values>
  <rowValues>
    <value>John</value>
    <value>Doe</value>
    <value><![CDATA[;-)]></value>
  </rowValues>
  <rowValues>
    <value>Smiley</value>
    <value>Face</value>
    <value><![CDATA[:-)]></value>
  </rowValues>
</values>
```

[field0]	[field1]	[field2]
John	Doe	;-)
Smiley	Face	:~)

Use each row once, or go around

By default the lines are used in turn, and after the last line the next call starts again at the first one. That is right for load tests: three accounts can make thousands of calls. Sometimes every line must be used exactly once: a list of real accounts to register or provision, numbers that must not be called twice, a list of devices to probe. Turn on **Options/Use Each Values Row Once** (- fvo on the command line): every line is given to one call, and after the last line the program makes no more calls. The run ends when those calls end, also when Total Calls is 0 (endless).

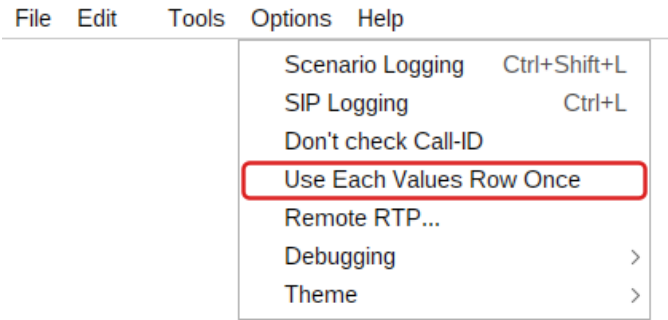


Figure 49 - Options/Use Each Values Row Once

Values file	Total Calls	Use Each Values Row Once	What happens
3 lines	8	off	8 calls with lines 1, 2, 3, 1, 2, 3, 1, 2
3 lines	0 (endless)	off	Calls without end, lines 1, 2, 3, 1, 2, 3 ...
3 lines	0 (endless)	on	3 calls, one per line, then the run is done
3 lines	8	on	3 calls: the lines end first
3 lines	2	on	2 calls (lines 1 and 2): Total Calls ends first

- The option is for the program as a client (it makes the calls). When the program is a server, an incoming call takes the next line; once every line is used the next calls get empty values, and SIPInspector.log says so.

- The option applies only when the scenario uses [fieldN] keywords; a scenario without them is not limited.
- Start on the progress screen begins again at the first line. The option is not kept after the program is closed, so a later test is never limited by surprise.

Counting values

A value written as *start+step* changes every time its line is used again, so one line can stand for thousands of different numbers. With the line below the first call uses [field0] = 5000000 and [field1] = 1, the second call 5000001 and 2, and so on:

5000000+1;1+1		
Written	Gives	Use
5000000+1	5000000, 5000001, 5000002 ...	a different caller number for every call
0099+1	0099, 0100, 0101 ... (leading zeros kept)	extensions or IDs of a fixed width
{100-5}	100, 95, 90 ...	counting down: write the value in braces
555-1234, +14155551234	always as written	phone numbers, dates and other text do not count

Counting continues across rounds. With two lines, a;1+1 and b;1+1, the calls get a1, b1, a2, b2, a3 ...

Example: values/Port_scan_values.txt has the single line 100;10024+1;: every call goes to the next port (10024, 10025, ...) of the address ending in .100, a port sweep with a one-line file.

Ready-made examples

Scenario	Values file	What it does
Reg_series_client.txt	Reg_series_values.txt	Registers a list of users with authentication: [field0] user, [field1] and [field2] authentication user and password.
PBX_call_series_client.txt	PBX_call_series_values.txt	Calls from many users to many numbers: [field0] caller, [field1] called number, [field2] and [field3] authentication.
Port_scan_client.txt	Port_scan_values.txt	Sends a REGISTER to a range of addresses and ports ([field0] last number of the address, [field1] the port).
any	values_incremental.txt	Two counting values: 5000000+1 and 1+1.
any	xmlValues.xml	An XML values file with a long XML body as a value.

The scenarios are in the folder `scenarios`, the values files in `values`.

When the values are not what you expect

What you see	Why, and what to do
An empty value where [field1] should be	The line has fewer values than the scenario uses, or no values file is loaded. SIPInspector.log has a warning that names the keyword. Check the number of ; on the line and File/Load Values.
All calls use the first line	The program was started again between the calls, or every run makes one call. Lines move forward with every call of one run.
Changes to the values file are not used	Press Start again: the file is read again when it was changed since the last run.
A number changes from call to call	It is written start+step (digits, +, digits). Write it differently, or it is meant as a counter.
Fewer calls than lines	Total Calls is lower than the number of lines, or the run was stopped. With Use Each Values Row Once set Total Calls to 0.

SIP message handling

Each SIP message has a direction and can be optional. If an incoming message is optional the control logic may skip to the next message and see whether that one matches. If that is optional too, it skips to the next, until it reaches a non-optional message. A non-optional incoming message must match: if it does not, the scenario does not advance. The idea behind optional messages is to handle a scenario that varies slightly from the expected sequence.

Outgoing messages can be optional too. It only makes sense after optional incoming messages: an optional outgoing message is sent only if the optional incoming message before it was received. For an outgoing call it is not known in advance if the server answers with 100, 180 or 200, so 100 and 180 are optional. A registration may be challenged with 401 or accepted at once with 200 OK:

```
Outgoing call
--> INVITE
<-- 100*
<-- 180*
<-- 200
--> ACK
---- Play pcap=test.pcap
<-- BYE
--> 200
```

```
Registration
--> REGISTER
<-- 100*
<-- 401*
--> REGISTER*
<-- 200
```

If an incoming message is mandatory the scenario will not advance until the expected message is received. Requests and responses are offered in a drop-down list, but you can always type a new message and add it to the scenario.

Messages that no call owns. A message whose Call-ID does not belong to a running call starts a new call only if the scenario has a dialog that begins with that message, for example a server scenario that begins with < - - INVITE, or a dialog after a New Dialog that waits for an incoming request. A dialog that begins with a message that SIP Inspector sends is started by SIP Inspector only. A late response or NOTIFY of an earlier call is counted as unexpected and is logged, and does not start anything. (Without this rule a client that subscribed on a real server could be flooded with calls created by the server's late NOTIFYs.)

Multiple dialogs in a scenario

Very often a single scenario has to handle several dialogs. Typical examples:

1. a client registers and receives a call;
2. a client registers and makes a call;
3. a server receives a registration followed by an incoming call;
4. a server receives a registration and makes a call;
5. a client registers every half an hour and handles incoming calls or OPTIONS messages.

A scenario with several dialogs must have a clear boundary between them: the **New Dialog** command (Command tab). Every dialog gets a Call-ID of its own.

```

a) Client registers and receives a call
---> REGISTER
<--- 407*
---> REGISTER*
<--- 200
New Dialog
<--- INVITE
---> 100
---> 180
---> 200
<--- ACK
---- Pause[ms]=15000
<--- BYE
---> 200

```

```

b) Client registers and makes a call
---> REGISTER
<--- 401
---> REGISTER
<--- 200
New Dialog
---> INVITE
<--- 100*
<--- 401
---> ACK
---> INVITE
<--- 100*
<--- 180
<--- 200
---> ACK
---- RTP Speech
---- Pause[ms]=15000
---> BYE
<--- 200

```

```

c) Server: registration, then a call
<--- REGISTER
---> 200
New Dialog
<--- INVITE
---> 100
---> 180
---> 200
<--- ACK
---- Pause[ms]=15000
<--- BYE
---> 200

```

```

d) Server: registration, then it calls
<--- REGISTER
---> 200
New Dialog
---> INVITE      // a new Call-ID is recommended
<--- 100*
<--- 180
<--- 200
---> ACK
---- Play pcap=test.pcap
---- Pause[ms]=15000
---> BYE
<--- 200

```

e) Periodic registration, incoming calls and OPTIONS:

```

start:
---> REGISTER
<--- 401
---> REGISTER
<--- 200
---- Pause[ms]=1800000
goto start
New Dialog
<--- INVITE
---> 100
---> 180
---> 200
<--- ACK
---- Play pcap=test.pcap
<--- BYE
---> 200
New Dialog
<--- OPTIONS
---> 200

```

Samples: Reg_call_client.txt and Reg_call_server.txt (a, b, c),
 Reg_subscribe_incoming_call_client.txt and PBX_reg_incoming_call_client.txt (e).

Option 'Don't check Call-ID'

SIP Inspector understands sessions: it places incoming and outgoing messages in the session (call) they belong to by the Call-ID, and creates a new session when the scenario says it should. In some cases it is more important to follow the order of the messages than to separate sessions. This is useful when the messages of different sessions are interleaved: a few messages of session 1, then a few of session 2, then again session 1, and so on, and all of them have to come in exactly that order. Tell the program to follow the order and not the sessions with **Options/Don't check Call-ID** (or `-noCID`).

With the option on:

- there is a single call. Every incoming message goes to it whatever its Call-ID is, and the scenario is executed strictly in the order of its entries;
- only one call runs at a time. Therefore **Calls/sec** and **Max Concurrent** are set to 1 and locked (they look different to show it). **Total Calls** still counts: the calls run one after another. When the option is turned off the fields get back the values they had before;
- turn the option on or off before pressing Run: the choice is read when the run starts.

Scenarios that are made from a capture with **Order: as captured** (see Trace to Scenario) need this option: they have the messages of all dialogs in one sequence, each dialog with a Call-ID of its own.

Unconditional RTP streaming

SIP Inspector can send RTP to a remote destination without any SIP signalling. Go to **Options/Remote RTP...**, fill in the address and the port, and create a scenario that has only a Play pcap entry and the needed pause. Press Run and the tool streams RTP to the remote RTP destination. It is useful to test how a SIP endpoint handles two RTP streams: the wanted one and the 'noise' it gets from an external and unwanted source. Make sure Calls/sec = 1 and Total Calls = 1. On the command line use `- rtp_ip` and `- rtp_port`.

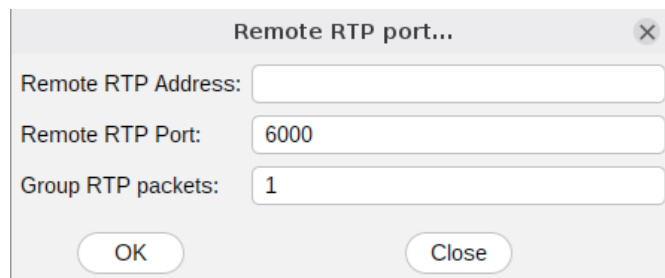


Figure 50 - Remote RTP

```
---- Play pcap=test.pcap
---- Pause[ms]=60000
```

Multipart SDP bodies

The program can send multipart bodies. There is an example in the scenarios folder: `HexBody_client.txt`. The main point is to create a boundary marker and use it to separate the bodies:

```

INVITE sip:[toUsername]@[remote_ip]:[remote_port];transport=[transport] SIP/2.0
Via: SIP/2.0/[transport] [local_ip]:[local_port];branch=[branch]ab
From: <sip:[fromUsername]@[remote_ip]:[remote_port]>;tag=[call_number]
To: <sip:[toUsername]@[remote_ip]:[remote_port]>
Call-ID: [call_number]@[local_ip]
CSeq: [cseq+1] INVITE
Max-Forwards: 70
Content-Type: multipart/mixed;boundary=level-0
Content-Length: [len]

--level-0
Content-Type: application/sdp
Content-Length: 324

v=0
o=MxSIP 131077 501465461 IN IP4 192.168.111.173
s=SIP Call
c=IN IP4 192.168.111.173
...

--level-0
Content-Disposition: signal; handling=required
Content-Type: application/qsig;version=iso
Content-Length: 324

v=0
...
--level-0--[no_\r\n]

```

The last delimiter ends with '--', which makes it the final one. For SIP Inspector not to add \r\n after the last delimiter, **[no_\r\n]** must be used.

Hexadecimal bodies

Hexadecimal content can be sent in a body. The content between the **[hex]** and **[/hex]** tags is read as a string of hexadecimal digits and the bytes are sent. If no \r\n must be added after the bytes, use **[no_\r\n]**:

```

INFO sip:[toUsername]@[remote_ip]:[remote_port];transport=[transport] SIP/2.0
...
Content-Type: application/qsig;version=iso
Content-Disposition: signal; handling=required
Content-Length: [len]

[hex]080200017b70028034[/hex][no_\r\n]

```

3rd Party Call Control

Setting a remote target makes it possible to talk to two or more different endpoints within the same scenario. In the example the program makes the first call, puts the remote side on hold, creates another dialog to a different endpoint and then transfers. If SIP Inspector talks to two or more endpoints it is not mandatory to use different Call-IDs; the INVITEs and their dialogs are told apart by where they are sent.

```

---- Remote=192.168.1.110:55634
--> INVITE          // first call
<-- 100*
<-- 180
<-- 200
--> ACK
---- Pause[ms]=2000
--> INVITE          // put it on hold
<-- 200
--> ACK
---- Pause[ms]=2000
---- Remote=192.168.1.105:5112
--> INVITE          // second call
<-- 100*
<-- 180
<-- 200
--> ACK
---- Pause[ms]=2000
---- Remote=192.168.1.110:55634
--> REFER
<-- 202
<-- NOTIFY
--> 200
---- Remote=192.168.1.105:5112
--> BYE
<-- 200
<-- BYE
--> 200

```

At the moment SDP is not passed between the calls automatically, so make sure that a codec that both endpoints can use is enforced. Sending SIP requests to different addresses can be combined with a values file: SIP requests are sent to a list of addresses, which lets you probe addresses and ports within a range (see `Port_scan_client.txt`). Complete transfer examples: `Call_transfer_client.txt`, `Call_transfer_server.txt` and `Call_brokering_b2bua_server.txt`.

Call generation

If the program works as a client the speed of call generation (calls per second), the maximum number of concurrent calls and the total number of calls can be set. A value of 10 for Calls/sec means create 10 calls every second; 0.1 means one call every 10 seconds. Max Concurrent = 0 means unlimited concurrent calls; Total Calls = 0 means unlimited. The logic:

1. new calls are generated at the Calls/sec speed, until
2. the maximum number of concurrent calls is reached: the program then waits for the number of calls to drop;
3. call generation continues until the total number of calls is reached.

The command line options are `-cps`, `-mc` and `-tc`. The special command `setCC(n)` and the keyword `[CCnt]` (the number of calls running now) change the load while a scenario runs: see `Load_gradual_increase_client.txt`.

IVR testing

With the ability to send digits the program is an easy way to test Interactive Voice Response systems and PBXs. The first scenario makes an outgoing call, establishes a two-way speech path, waits 9 seconds, dials a random 7-digit number and waits 11 seconds so all digits are sent before it hangs up. The second one dials fixed digits:

```

---> INVITE
<--- 100*
<--- 180
<--- 200
---> ACK
---- RTP Speech
---- Pause[ms]=9000
---- rndVal=rand[1000000-9999999]
---- Dial=[$rndVal]
---- Pause[ms]=11000
---> BYE
<--- 200

```

```

---> INVITE
<--- 100*
<--- 180
<--- 200
---> ACK
---- RTP Speech
---- Pause[ms]=9000
---- Dial=123456;PT=101
---- Pause[ms]=11000
---> BYE
<--- 200

```

Besides random values the digits can come from the values file ([field0]) or be auto-incremented. The sample is `Call_speech_dials_client.txt` (it needs a sound device because it uses live speech).

Trace to Scenario

Writing a scenario by hand is slow when the traffic is complex. **Trace to Scenario** reads a Wireshark capture (pcap or pcapng) and makes a scenario from the SIP messages of **one endpoint**. The endpoint is the side that SIP Inspector takes the place of: what it sent becomes a message to send, what it received becomes a message to wait for. The RTP that the endpoint sent can be written to pcap files that the scenario plays. In the GUI it is **Tools/Trace to Scenario**, on the command line it is `-pcap2scen`.

A typical use: a phone or a PBX behaves badly, you capture its traffic with Wireshark, convert the capture into a scenario of the phone, and replay it against the system under test whenever you need, with different addresses and numbers. Or the other way around: convert the capture into a scenario of the *server* and test a phone against it.

Makes a scenario from the SIP messages of one endpoint of a Wireshark capture: what it sent is sent, what it received is waited for.

Capture file: Browse...

6993 frames, 76 SIP messages, 2 RTP streams

Who is SIP Inspector? Select the endpoint (MAC address, or IP address if the capture has none).

Endpoint (MAC or IP)	IP addresses	SIP sent	SIP received	RTP streams sent
48:3a:02:92:00:63	192.168.2.102	38	38	1
68:69:2e:03:66:10	192.168.2.78	38	37	1
01:00:5e:00:01:4b	224.0.1.75	0	1	0

Dialogs (one per Call-ID): tick those for the scenario. Keep Call-ID writes the Call-ID of a dialog that the endpoint started as captured, not as [call_number].

Use	Keep Cal...	Start (s)	Role	Dialog	Outcome	Messages	Length (s)	Notes
☐	☐	0.00	client	REGISTER uG0ME3mstTMWHAui9O...	REGISTER -> 200	4	0.09	authentication, overlaps 1 other dialog, with 192.168.2.78
☐	☐	0.02	client	SUBSCRIBE RPYBsGULaelUuQHNG...	SUBSCRIBE -> 200	4	0.01	overlaps 1 other dialog, with 224.0.1.75
☐	☐	0.09	client	SUBSCRIBE 8FnYK.DBJ5UykbCWy...	SUBSCRIBE -> 200	6	0.15	authentication, overlaps 4 other dialogs, with 192.168.2.78
☐	☐	0.10	client	SUBSCRIBE HcaOcaNnN6obePknV1...	SUBSCRIBE -> 200	16	77.18	authentication, overlaps 9 other dialogs, with 192.168.2.78
☐	☐	0.11	client	SUBSCRIBE HgRoXIML19gKrzjW3xv...	SUBSCRIBE -> 200	6	0.17	authentication, overlaps 4 other dialogs, with 192.168.2.78

Type: from the first dialog Calls Registrations Subscriptions Complete All None

Order: by dialog (for Check Call-ID) Accept a NOTIFY before or after the response to its SUBSCRIBE

Keep the timing of the capture exactly 500 ms

Wait before it	Step of the endpoint	Wait in the capture (ms)
☐	---> INVITE	0
☒	---> ACK	3
☒	---> INVITE	2

Tick the steps that wait as in the capture. All None Reset Write the RTP it sent to pcap files and play them Use [fromUsername] and [toUsername] instead of the numbers of the capture

Scenario file: Browse...

RTP pcap folder: Browse...

Close Convert

Figure 51 - Tools/Trace to Scenario, with a capture of a Fortinet phone loaded (the whole window; the next two figures show its parts larger)

Makes a scenario from the SIP messages of one endpoint of a Wireshark capture: what it sent is sent, what it received is waited for.

Capture file

6993 frames, 76 SIP messages, 2 RTP streams

Who is SIP Inspector? Select the endpoint (MAC address, or IP address if the capture has none).

Endpoint (MAC or IP)	IP addresses	SIP sent	SIP received
48:3a:02:92:00:63	192.168.2.102	38	38
68:69:2e:03:66:10	192.168.2.78	38	37
01:00:5e:00:01:4b	224.0.1.75	0	1

Dialogs (one per Call-ID): tick those for the scenario. Keep Call-ID writes the Call-ID of a dialog that the endpoint started as captured, not as [call_number].

Use	Keep Cal...	Start (s)	Role	Dialog	Outcome	Messages	Length (s)	
☐	☐	0.00	client	REGISTER uGOME3mstTMWHauil9O...	REGISTER -> 200	4	0.09	authent
☐	☐	0.02	client	SUBSCRIBE RPYBsGULaelUhQIHNG...	SUBSCRIBE -> 200	4	0.01	overlap
☐	☐	0.09	client	SUBSCRIBE 8FnYK.DBj5IJYkbCWy...	SUBSCRIBE -> 200	6	0.15	authent
☐	☐	0.10	client	SUBSCRIBE HcaOcaNnN6obePknV1...	SUBSCRIBE -> 200	16	77.18	authent
☐	☐	0.11	client	SUBSCRIBE HqRoXIML19gKrzjW3xv...	SUBSCRIBE -> 200	6	0.17	authent

Type: from the first dialog ▾

Calls Registrations Subscriptions Complete

Figure 52 - The endpoints and the dialogs of the capture: Use, Keep Call-ID, role, outcome (the Notes column is further to the right)

Type: from the first dialog ▾

Order: by dialog (for Check Call-ID) ▾ ☒ Accept a NOTIFY before or after the response to its SUBSCRIBE

Keep the timing of the capture exactly ▾ 500 ms

Wait before it Step of the endpoint

☐	---> INVITE
☒	---> ACK
☐	---> INVITE

Tick the steps that wait as in the capture. All None Reset ☒ Write the RTP it sent to pcap files and play them ☐ Use [fromUsername] and [toUser]

Scenario file

RTP pcap folder

Figure 53 - Type, order, timing, RTP, numbers and the files to write

Step by step in the GUI

- Capture file.** Press **Browse...** and pick a pcap, pcapng or cap file. The program reads it and prints how many frames, SIP messages and RTP streams it has. Retransmitted messages are ignored. Frames in Ethernet, VLAN-tagged, Linux cooked, IPv4 and IPv6, UDP and TCP (including messages that arrive in several TCP segments or IP fragments) are understood. SIP over TLS or WebSocket is encrypted and cannot be read.
- Who is SIP Inspector?** The table lists the endpoints that take part in SIP, the most active one first: the MAC address (the IP address if the capture has no MAC addresses), the IP addresses, how many SIP messages it sent and received and how many RTP streams it sent. Select one.
- Dialogs.** A dialog is all the messages with one Call-ID: a registration, a call, a subscription. The table lists the dialogs of the selected endpoint with the start time, the role (*client* if the endpoint sent the first message, *server* if it received it), the first message and the Call-ID, the outcome (for example REGISTER -> 200), the number of messages, the length in seconds, and notes (authentication, overlaps with other dialogs, the other side, a message that is cut off, a dialog that started before the capture). Tick **Use** for the dialogs that go into the scenario. The buttons **Calls**, **Registrations**, **Subscriptions**, **Complete** (every dialog that starts in the capture), **All** and **None** tick them quickly. The program starts with the calls ticked (or, if there are none, every dialog that starts in the capture).
- Keep Call-ID.** By default the Call-ID of every dialog becomes the keyword [call_number], so every run has new Call-IDs. Tick **Keep Call-ID** for a dialog to write the Call-ID exactly as it was captured. The choice is made for each dialog, and only for dialogs that the endpoint started (role client). Use it when the system under test must see the original Call-ID. A scenario that keeps a Call-ID should run one call only (Total Calls = 1): a second call would repeat the same Call-ID.
- Type.** *Client*: the endpoint starts the dialogs. *Server*: the endpoint answers; each dialog starts with a message it received. *From the first dialog*: whichever the first ticked dialog is; ticked dialogs of the other kind are left out. A scenario cannot be both.

6. **Order.** *By dialog (for Check Call-ID):* the messages of a dialog stay together and the dialogs follow each other in the order they started, every dialog a call of its own; the messages of different dialogs do not have to come in the captured order. *As captured (for Don't check Call-ID):* one sequence with the messages of all dialogs mixed as they were, every dialog with a Call-ID of its own; run it with Options/Don't check Call-ID, where the order is strict. **Accept a NOTIFY before or after the response to its SUBSCRIBE** (order by dialog only) makes the scenario take both orders, as real servers send them in either order.
7. **Timing.** What happens with the time between the messages: *Keep the timing of the capture exactly*, *Keep only waits of at least ... ms*, or *No timing: send as fast as possible*. The table below lists every step of the endpoint with the wait before it in the capture; a tick means 'wait this long'. Change single steps by hand; **All**, **None** and **Reset** set all of them at once or go back to what the timing mode says.
8. **Write the RTP ...** writes the RTP that the endpoint sent into pcap files and puts a Play pcap entry for each in the scenario. **Use [fromUsername] and [toUsername] ...** writes the number that calls and the number that is called as keywords, so the scenario uses what is entered in the From and To of the Account section; unticked, the numbers stay as captured.
9. **Scenario file** and **RTP pcap folder** say where the files go. Press **Convert**. The box at the bottom lists the files that were written, how many messages are sent and waited for, and warnings. Load the result with File/Load Scenario. Files that exist are replaced only after you confirm.

What the scenario looks like

Everything that belongs to one run of a call is replaced with a keyword: addresses, ports, tags, Call-ID, branch, CSeq and the media port. The scenario therefore does not depend on the capture. Headers that identify the device (User-Agent, Supported, Allow, ...) stay as captured. A comment at the top of the scenario tells where it comes from, what it does and what to keep in mind:

```
// Client: made from the capture pcap_trace.pcap
//
// Endpoint 48:3a:02:92:00:63 (192.168.2.102), the other side is 192.168.2.78:5060, UDP, 1 dialog
// Addresses, ports, tags, Call-ID, branch, CSeq and the media port are keywords, so the scenario
// does not depend on them.
// WARNING: Run SIP Inspector as a client with the remote address 192.168.2.78:5060 (or the address
// of the system under test).
//
// SIP Messages
// dialog INVITE Ecxpww6c8TmEBT.j0tnx1LGH... INVITE -> 200
// ----> INVITE
// <--- 401 Unauthorized
// ---- pause 3 ms
// ----> ACK
// ----> INVITE
// <--- 100 Trying (optional)
// <--- 180 Ringing (optional)
// <--- 200 OK
// ---- play call_from_trace_rtp1_audio.pcap
// ----> ACK
// <--- BYE
// ----> 200 OK
```

Details that the program takes care of:

- **Authentication.** A request that answers a 401 or 407 challenge gets Authorization:
[authentication username=9040; password=[authPassword];]: the user name of the capture and the password that is set in SIP Inspector (the captured digest response cannot be reused, it belongs to the nonce of the capture). Set the password in 'Auth password' or with -ap.
- **Variables.** A tag, a Call-ID or a Contact that the other side made and that the endpoint repeats later is read from the received message into a variable, not copied.
- **Responses** of the endpoint copy Via, From, To, Call-ID and CSeq of the request they answer (the [last_...] keywords).
- **Optional messages.** 100 Trying and 180 Ringing are written as optional messages, since a server may or may not send them.

- **Other destinations.** A message that goes to another address than the main peer (for example a multicast address) is preceded by Set Remote.
- **SDP.** The media port is [media_port] (or [video_port]), the address is [local_ip_sdp]; the a=rtcp lines are left out and a warning says so.
- **Overlapping dialogs.** A scenario runs one dialog after the other, so dialogs that overlapped in the capture are written one after the other. A warning says how many.

Playing it

Load the scenario, enter the address of the other side as the Remote Server (the scenario comment tells which address the capture had), the From and To, and the authentication password, and press Run. A scenario made with *Order: as captured* needs Options/Don't check Call-ID.

On the command line: -pcap2scen

-pcap2scen is a command of its own. It does not start the GUI, needs no license and does not touch the settings. It does what the window does, with options:

Option	Meaning
-ep <ip mac>	The endpoint to make the scenario for (default: the one with the most SIP messages).
-fs <file>	The scenario file to write (default: scenarios/<name of the capture>.txt, or ./<name>.txt when there is no scenarios folder).
-type <t>	client, server or auto: what the first selected dialog is (default auto).
-order <o>	dialog: the messages of a dialog together, for Check Call-ID (default); capture: one sequence in the order of the capture, for -noCID.
-timing <t>	exact: keep every wait of the capture (default); none: no waits; <ms>: keep the waits of at least that long.
-calls <ids>	The dialogs to take, Call-IDs (or the beginning of them) separated by commas, or all (default: the calls if there are any, else every dialog that starts in the capture).
-noreorder	Do not accept a NOTIFY before the response to its SUBSCRIBE.
-numbers <n>	keep: the numbers of From and To as in the capture (default); keywords: [fromUsername] and [toUsername].
-nortp	Do not write the RTP of the endpoint to pcap files.
-pcapdir <dir>	The folder for the RTP pcap files (default: pcap).
-list	Only list the endpoints and the dialogs of the endpoint, write nothing.
-y	Replace files that exist (without it the command stops).
-h	Print this help.

First look at what is in the capture with -list. The endpoint that has a * is the one that would be used:

```

Terminal
$ SIPInspDlx -pcap2scen pcap_trace.pcap -list
pcap_trace.pcap: 6993 frames, 76 SIP messages, 2 RTP streams

Endpoint          Addresses          SIP sent SIP rcvd   RTP
* 48:3a:02:92:00:63 192.168.2.102      38       38       1
68:69:2e:03:66:10 192.168.2.78       38       37       1
01:00:5e:00:01:4b 224.0.1.75         0        1        0

Endpoint: 48:3a:02:92:00:63 (192.168.2.102) (the one with the most SIP messages; -ep chooses another)

start msgs  role  Call-ID          first message / outcome
0.00 4      client  uG0ME3mstTMWHauI90fVxjtq1NbW.aA  REGISTER REGISTER -> 200
0.02 4      client  RPYBsGULaelUhQlHNGZRnRXRH-LdUkXA  SUBSCRIBE SUBSCRIBE -> 200
0.09 6      client  8FnYK.DBJj51JYkbCwyFtcRTd0XpJkqz  SUBSCRIBE SUBSCRIBE -> 200
0.10 16     client  Hca0caNnN6obePknV1gZnmbSi.3lf0pA  SUBSCRIBE SUBSCRIBE -> 200
0.11 6      client  HqRoXiML19gKrZjjW3xvPm945ao45BJH  SUBSCRIBE SUBSCRIBE -> 200
0.11 16     client  k-po1brEhqIw5Z3iOud5G9d9kpt0lbUo  SUBSCRIBE SUBSCRIBE -> 200
0.12 6      client  amrcG62NEuYk1ImeM0f6mndDK0axnB0H  SUBSCRIBE SUBSCRIBE -> 200
12.38 10     client  EcxpUw6c8TmEBT.j0tnx1LGHVB5.uj8S  INVITE INVITE -> 200
12.68 2      server  PRfIsM9EemT43ppyzvYUJz5lXWkuX7T2  NOTIFY NOTIFY -> 481 (started before
the capture)
12.70 2      server  JPiKLy7.vjkiQ25SCfcmDRKt9pCpmxBW  NOTIFY NOTIFY -> 481 (started before
the capture)
13.50 2      server  KcmTUwEcIHfPERmTufhqfe9j0cxij-nf  NOTIFY NOTIFY -> 481 (started before
the capture)
13.50 2      server  EhmG-sbZfYiytA03PwN-MuZaXdq-sEbY  NOTIFY NOTIFY -> 481 (started before
the capture)

```

Figure 54 - Listing the endpoints and the dialogs of a capture

Then convert. This makes a client scenario from the INVITE dialog only (the Call-ID is given by its beginning), writes the RTP that the phone sent into the folder pcap_out, and replaces the files if they exist:

```

Terminal
$ SIPInspDlx -pcap2scen pcap_trace.pcap -type client -calls EcxpUw6c -fs scenarios/call_from_trace.txt -y
pcap_trace.pcap: 6993 frames, 76 SIP messages, 2 RTP streams
Endpoint: 48:3a:02:92:00:63 (192.168.2.102) (the one with the most SIP messages; -ep chooses another)
Scenario written: scenarios_out/call_from_trace.txt (1 dialog, 5 messages to send, 5 messages to wait for)
RTP pcap written: pcap_out/call_from_trace_rtp1_audio.pcap (3096 packets)
Run it as a client, with the remote address of the other side (192.168.2.78:5060 in the capture).
WARNING: a=rtcp lines of the SDP were left out: they gave the RTCP port of the capture, and the media port
is a keyword.
WARNING: Run SIP Inspector as a client with the remote address 192.168.2.78:5060 (or the address of the
system under test).
The scenario answers authentication challenges with the user name of the capture and the password given
with -ap (the captured digest response cannot be reused).

```

Figure 55 - Converting one dialog of the capture

Exit codes of -pcap2scen: 0 - the scenario was written (or the list was printed); 1 - the conversion failed (no such capture, no SIP in it, a file that exists and -y was not given, a dialog that is not in the capture); 2 - the command line is wrong.

Note: Keep Call-ID can be chosen only in the GUI.

Writing scenarios with Claude

Claude, the AI assistant of Anthropic, can write a scenario from a description of the call flow, for example *a client that registers with authentication, calls 2000, sends the digits 1234 and hangs up after 20 seconds*. The program comes with a **skill** that teaches Claude the scenario format, the keywords and the sample scenarios. With it Claude starts from the sample closest to the request, writes the scenario and its header comment, checks it, and when it is asked for both sides, plays them against each other on this computer before it hands them over. Claude is a separate product: it needs an account at claude.ai, and SIP Inspector does not send anything to it.

Where	What is installed	How to use it
Claude Code (command line)	<code>.claude/skills/sip-inspector-scenarios</code> in the program folder	Open a terminal in the program folder, start <code>claude</code> and describe the scenario. The skill is found by itself.
claude.ai (browser, desktop app)	<code>sip-inspector-scenarios.skill</code> in the program folder	Settings/Capabilities: turn on code execution, then upload the file under Skills. Attach the closest sample scenario to the chat.

In Claude Code the skill uses the folder of the program: it reads the samples in `scenarios`, saves the new scenario there and can run the CLI to test it, so the program folder must be one that the user can write to. In `claude.ai` Claude cannot see the program, so it checks the scenario but cannot run it; run it with the CLI or the GUI.

The skill has two scripts that can also be used without Claude (Python 3, nothing else to install):

```
python3 .claude/skills/sip-inspector-scenarios/scripts/check_scenario.py scenarios/my_call.txt
python3 .claude/skills/sip-inspector-scenarios/scripts/loopback_test.py --root . \
--server Reg_call_server.txt --client Reg_call_client.txt
```

- **check_scenario.py** reads a scenario as the program does and reports what would be ignored or refused (a block with two commands, a goto without its label, an assertion that cannot be read) and what the other side would likely reject (no tag in From, a CSeq with the wrong method, a body without Content-Length: [len], an SDP without an a=rtpmap). `--outline` prints the call flow.
- **loopback_test.py** starts the server scenario, plays the client scenario against it on 127.0.0.1 and reports PASS or FAIL for both sides. It needs the registered CLI. `--pause-ms 2000` shortens the pauses for a quick run.

Tip: A scenario that Claude wrote is a scenario like any other: read its header comment, run it against a test system first, and check the run report.

Public/private keys

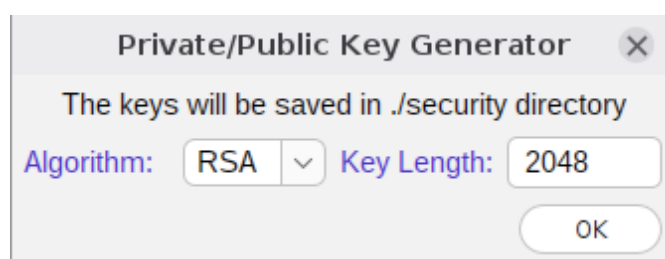


Figure 56 - Tools/Security Keys

Keys are easily generated by SIP Inspector (**Tools/Security Keys**, or `-genKeys`). Four types are offered: RSA, DSA, DH and EC. The key length is in bits. The general rule is: the more bits, the stronger the encryption and the slower the processing. EC (elliptic curve) keys are the best of both worlds: they are relatively short and give superb security and speed. The length is checked: RSA, DSA and DH keys are 512 to 16384 bits; EC lengths are the curve sizes 163, 192, 224, 233, 256, 409 and 571. To generate keys select the algorithm and the length,

press OK, and the keys are in the security folder (`private.pem` and `public.pem`). The table compares RSA and EC lengths of about the same strength:

RSA	EC (supported sizes)
2048 bits	224 bits
3072 bits	256 bits
7680 bits	409 bits
15360 bits	571 bits

Self-signed certificate



Figure 57 - Tools/Security Certificate

To create a self-signed certificate use **Tools/Security Certificate** (or `-genCrt`). The screen is pre-populated. The important part is the **Common Name**: it must be the FQDN or the IP address of the server that the certificate is made for. The keys must exist first: generate them with the tool above. Pressing **Generate Certificate** and then **OK** creates `certificate.crt` in the security folder and puts the key and the certificate into the keystore. The program calls the `keytool` program that comes with Java for this, so `keytool` must be found on the `PATH`.

The program checks what it is given: a certificate that does not match its private key is refused and no keystore is made from it, and a truststore that is damaged is left as it is instead of being overwritten.

SDP calculator

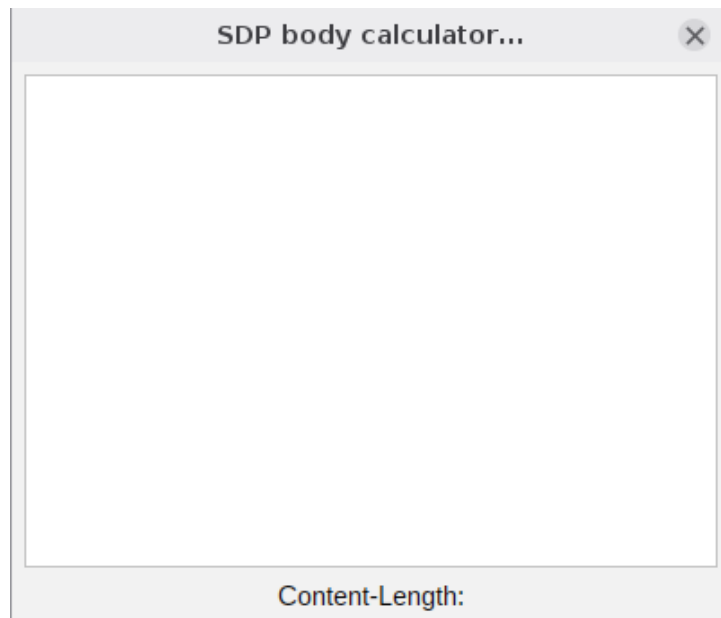


Figure 58 - Tools/SDP Calculator

A SIP message with a body needs the Content-Length header. In a scenario the keyword **[len]** is replaced with the length of the body when the message is sent, so it is usually not needed. When you must know the length yourself, for example to build a message by hand, paste the body into the calculator and it shows the Content-Length.

Logging and troubleshooting

There are three different logs. Use the one that answers your question:

Log	Contents	How to turn it on	Where
SIP messages	Every SIP message sent and received, as it is on the wire.	Options/SIP Logging (Ctrl+L), or -log	Logs/<date and time>.log
Scenario progress	The progress table of the scenario screen (the Exp and Unexp counters).	Options/Scenario Logging, or -logScen	Logs/<scenario>_<date and time>.log
Program log	What the program does: the steps of a run, of every call, warnings and errors. This is the one to send with a bug report.	Always on at DEBUG; change the level with Options/Debugging or -d	SIPInspector.log in the folder the program was started in

The levels of the program log: **ERROR** - what stops a call or a feature from working; **WARN** - something unexpected that the program survived; **DEBUG** - the steps of a call, a connection or a scenario that are needed to follow what happened (the default); **VERBOSE** - the detail of every packet. In the Debugging menu they are called Error, Warning, Info, Fine and Off. The log is limited to 20 MB.

When the counters are wrong

- **The scenario does not start.** Check the local address and port: another program may use the port, or the address is not the one of your network adapter. Run the CLI with -d DEBUG and read the first lines of SIPInspector.log.
- **Nothing comes back.** Check the remote address and port and any firewall. Turn on SIP Logging: if only outgoing messages are in the log, the other side does not answer.
- **Many unexpected messages.** Look at the Unexp msgs column to see what came and at which entry. The program log has a line for each one, for example 'received 401 that the scenario does not expect now; the scenario is at entry 0'. Typical reasons: an optional message is missing in the scenario, the other side repeats messages (retransmissions), or the other side answers with a challenge (401, 407) that the scenario

does not handle.

- **Messages of one call end up in another.** The scenario has several dialogs that interleave: use Options/Don't check Call-ID.
- **A call waits forever.** A mandatory incoming message does not come, or Wait RTP waits for audio that is not sent. Use the Pause command and Wait video's time limit, and the -to option in the CLI.
- **For a bug report** send the diagnostics bundle (Help/Save Diagnostics or -diag): it has SIPInspector.log, the scenario file, the settings and the call flows.

The diagnostics bundle

Help/Save Diagnostics saves one zip with what is needed to look at a problem, so a support question starts with the facts. Send it to support@sipinspector.com with a few words on what you did and what went wrong.

File	What it has
about.txt	The version, the license state and the Machine-ID, Java, the operating system, the memory, the network cards, the log level, the command line
settings.txt	Every setting of the run
scenario/, scenario-check.txt	The scenario file, and what Validate Scenario finds in it
logs/	SIPInspector.log (its last 8 MB) and the two newest SIP message logs
reports/	The three newest reports of the Reports folder
run/	The report of the last run (HTML and JSON), what the SIP/SDP checks found with the messages, and the call flows as text
values/	The values file, only when it is asked for: it often has the passwords of many users

Passwords are not in it. The password and the API token of the settings, the serial number, a password= of a scenario and the digest response of the Authorization headers are replaced with *** in every file. Look through the bundle before you send it if your scenario or log has other things that must not leave your network, such as addresses or user names.

On the command line, -diag bundle.zip writes it when the run ended, after the reports; a bundle that cannot be written never changes the exit code. Without -fs and -api, SIPInspDlx -diag bundle.zip writes it at once: it works when the program cannot start a run, even when the license is the problem. The API gives it with GET /diag.

Sample scenarios

The folder scenarios has many ready-to-run scenarios. Most come in pairs: a *client* scenario and the *server* scenario that answers it, so you can run both with SIP Inspector, one against the other, on one computer. Every file starts with a comment that tells what it does.

Scenario	What it does
Asterisk_reg_client.txt	Client: registers with an Asterisk server
Asterisk_reg_incoming_call_client.txt	Client: registers with Asterisk and answers incoming calls
Asterisk_reg_outgoing_call_client.txt	Client: registers with Asterisk and makes a call
Attended_transfer_pbx_client.txt	Client: attended call transfer, as a transferor (lab scenario)
Attended_transfer_phones_client.txt	Client: attended call transfer, as a transferor (lab scenario)
Attended_transfer_target_server.txt	Server: answers a call and then calls extension 109 (lab scenario)

Scenario	What it does
Call_brokering_b2bua_server.txt	Server: registrar and call broker (back-to-back user agent)
Call_busy_client.txt	Client: call that is rejected with 486 Busy Here
Call_busy_server.txt	Server: rejects every call with 486 Busy Here
Call_cancel_client.txt	Client: call that is cancelled while it rings (CANCEL, 487 Request Terminated)
Call_cancel_server.txt	Server: rings and accepts the CANCEL of the caller
Call_codec_change_rtp_only_client.txt	Client: call whose RTP changes in mid-call without any signalling
Call_codec_change_signaling_client.txt	Client: call with a mid-call codec change by re-INVITE
Call_codec_change_signaling_server.txt	Server: answers a call and a re-INVITE (mid-call change)
Call_hold_resume_client.txt	Client: call put on hold and taken off hold with re-INVITES
Call_hold_resume_server.txt	Server: answers a call that the caller puts on hold and resumes
Call_speech_dials_asserts_client.txt	Client: call with live speech and dialled digits, with pass/fail assertions
Call_speech_dials_client.txt	Client: call with live speech and dialled digits
Call_speech_dials_server.txt	Server: answers one incoming call
Call_transfer_client.txt	Client: transfers a call (blind transfer, RFC 3515), playing two roles
Call_transfer_server.txt	Server: the transferee of a blind transfer (RFC 3515)
Call_video_client.txt	Client: video call, H.264 and PCMU
Call_video_server.txt	Server: video call, H.264 and PCMU
HexBody_client.txt	Client: call with binary (QSIG) bodies written in hexadecimal
HexBody_server.txt	Server: call with binary (QSIG) bodies written in hexadecimal
Load_gradual_increase_client.txt	Client: raises the number of simultaneous calls step by step
Message_client.txt	Client: SIP instant message (MESSAGE, RFC 3428)
Message_server.txt	Server: receives a SIP instant message (MESSAGE) and answers 200 OK
Options_ping_client.txt	Client: OPTIONS ping - is the server up and answering SIP?
Options_ping_server.txt	Server: answers OPTIONS with 200 OK and its capabilities
PBX_call_codec_change_reinvite_client.txt	Client: call through a proxy with a codec change by re-INVITE
PBX_call_series_client.txt	Client: outgoing calls, one per row of a values file
PBX_reg_incoming_call_client.txt	Client: registers every minute and answers incoming calls
Perf_cps_client.txt	Client: call rate test - a short call without media
Perf_cps_server.txt	Server: call rate test - answers a short call without media
Perf_rtp_client.txt	Client: RTP test - a call with G.711 audio both ways
Perf_rtp_server.txt	Server: RTP test - answers and sends G.711 audio until the BYE
Port_scan_client.txt	Client: sends a REGISTER to many addresses and ports
PRACK_client.txt	Client: call with reliable provisional responses (RFC 3262)
PRACK_server.txt	Server: answers a call with a reliable 180 Ringing (RFC 3262)
Record_route_client.txt	Client: call through proxies (Record-Route / Route)
Record_route_server.txt	Server: call through proxies (Record-Route / Route)
Reg_auth_call_advanced_client.txt	Client: registers and calls, following the server's challenge
Reg_auth_call_client.txt	Client: registers and calls, both challenged

Scenario	What it does
Reg_auth_call_server.txt	Server: registrar and call server that challenges the REGISTER and the INVITE
Reg_branching_client.txt	Client: registers with authentication, branching on the response
Reg_call_client.txt	Client: registers, then makes a call
Reg_call_server.txt	Server: registrar that also takes a call
Reg_client.txt	Client: registers with authentication
Reg_series_client.txt	Client: registers many users, one per row of a values file
Reg_server.txt	Registrar server with authentication scenario
Reg_subscribe_call_client.txt	Client: registers, subscribes for message-waiting notifications, then makes a call
Reg_subscribe_call_server.txt	Server: registrar, message-waiting notifier and call server
Reg_subscribe_incoming_call_client.txt	Client: registers, subscribes for message-waiting notifications, then receives a call
Reg_subscribe_incoming_call_server.txt	Server: registrar and message-waiting notifier that calls the client
SRTP_reg_incoming_call_client.txt	Client: registers and answers an SRTP call
SRTP_reg_outgoing_call_client.txt	Client: SRTP call to a server that challenges with 401
SUBSCRIBE_dialog_client.txt	Client: subscribes to dialog events (BLF)
SUBSCRIBE_dialog_server.txt	SUBSCRIBE server scenario - subscribes for dialog event (BLF) Subscriber username: any Subscribing to: any Scenario simulates subscription, and a full cycle state changing: terminated->early->confirmed->terminated.
WebRTC_reg_incoming_call_server.txt	Server: WebSocket server that takes a call from a browser (WebRTC)
WebRTC_reg_outgoing_call_server.txt	Server: WebSocket server that calls a registered browser (WebRTC)

Testing the program

The test folder has two scripts that you can run from the root folder of the source (the one that has `src`, `lib` and `scenarios`). You need a JDK; `scenario_test.py` needs Python 3.6 or newer.

- **test/run.sh** - the unit tests of the program code. It compiles the sources into a temporary folder and runs the tests one after the other. Every check prints PASS or FAIL. They cover, among other things, sending RTP to several clients, SRTP, keyword replacement, scenario comments, H.264 video, the serial number and the Machine-ID, the security tools, Trace to Scenario and the Command tab. Run `test/run.sh bench` for the speed test of the keyword replacement.
- **test/scenario_test.py** - runs scenario files against each other. A server scenario and a client scenario that belong together (`test/pairs.txt` lists the pairs) are started with the real program, the client makes the calls, and the result is judged. Options let you set how many calls, how many at the same time and at what speed: `test/scenario_test.py -n 1000 -c 50 --cps 100` Reg makes 1000 registrations, 50 at a time, 100 per second. This is a good way to check that a new scenario has its counterpart and to load-test.

Version checker

Occasionally a new version of SIP Inspector becomes available. The program checks for it when it starts. The check runs in the background, so the main window is shown at once and it never waits for the network. If a newer version exists a window offers to download it; sometimes a message from the author is shown. Licenses bought for version 7 allow downloading any update of 7.xx for free. Updates are usually bug fixes or smaller features that make the program faster, better and more functional.

Keywords

Before a message is sent, every keyword in it is replaced with its value. Keywords are written in square brackets. To send a real '[' (for example in an IPv6 address) write it twice: `[[`.

Keyword	Replaced with
[authentication]	A complex keyword that takes a user name and a password: [authentication username=value1; password=value2;]. It is replaced with an Authorization (or Proxy-Authorization) header with the MD5 digest calculated for that user name and password. If the previous message is not a 401 or 407 nothing happens. The values may be keywords: username=[authUsername]; password=[authPassword];.
[authUsername]	The authentication user name of the Account section (or -au).
[authPassword]	The authentication password of the Account section (or -ap).
[branch]	The branch parameter of the Via header: "z9hG4bK", the call counter and the position of the entry in the scenario. For example z9hG4bK15.
[call_number]	The global call counter. It grows with every new call. A new dialog of the same call gets a new value too, as the suffix after a dash.
[CCnt]	The number of calls that run now.
[cseq]	The current CSeq value of this call.
[cseq+N], [cseq-N]	Adds N to (subtracts N from) the CSeq value of the call and is replaced with the new value. The next [cseq] has the new value too. An INVITE written with [cseq+1] and the ACK with [cseq] have the same number.
[IP_ver]	IP4 or IP6, as the IP version of the initial screen says. Used in the SDP.
[fieldN]	A value of the values file. N is the 0-based index of the column. The first call reads the first row, the second call the second row, and so on (see Values files: one scenario, many users).
[fromUsername]	The From of the Account section (or -uf).
[toUsername]	The To of the Account section (or -ut).
[last_*:]	Used to build responses. Replaced with the header(s) of that name of the last received message; * is the header name. [last_Via:] gives all Via headers of the previous message.
[val_last_*:]	The <i>value</i> of a header of the last received message, without the name. For example RACK: [val_last_RSeq:] [val_last_CSeq:] when working with PRACK.
[len]	The length of the body of the message in bytes.
[local_ip]	The local IP address. An IPv6 address is in square brackets. (-lip)
[local_ip_sdp]	The local IP address for the SDP: an IPv6 address is not in square brackets.
[local_port]	The local signalling port the message is sent from (-lp). For TCP and TLS, where a client uses an ephemeral port, it is the port that is really used.
[media_port]	The RTP port of the RTP field of the Account section: where RTP is received and sent from (-lrtp).
[video_port]	The port that the program opened for the video of this call. Used in the video line of the SDP.
[peer_tag_param]	The From tag, if the previously received message is a request, or the To tag, if it is a response; empty when there is none.

Keyword	Replaced with
[remote_ip]	The IP address of the remote server or, in server mode, of the address the last message came from. An IPv6 address is in square brackets. (- r ip)
[remote_port]	The remote server port or, in server mode, the port the last message came from (- rp).
[transport]	UDP, TCP, WS, WSS or TLS, as selected.
[\$var]	The value of the user variable 'var' (see Variables).
[\$var+N], [\$var-N]	The value of 'var'; the variable is then increased (decreased) by N, so the next reference gets the new value.
[ver]	The SIP Inspector version.
[hex]...[/hex]	Hexadecimal digits in a body, sent as bytes.
[no_\r\n]	At the end of a body: do not add \r\n after it.
[[	A literal '['.

Tip: A keyword that the program does not know is left in the message as it is. If a message shows a keyword like [fromUsername] on the wire, check the spelling.

Performance and capacity

How many calls a computer can make or answer with SIP Inspector depends on the computer: the processor above all, and the network. This chapter shows what was measured on real computers and how to measure your own with the same method: `test/capacity.py` of the source distribution and the scenarios `Perf_cps_client/server` and `Perf_rtp_client/server`.

What is measured

- **Calls per second** - short calls without media: INVITE, 100, 180, 200, ACK, one second, BYE, 200. The call rate is raised step by step, each step a minute of calls, until a step fails its limits. This is the signalling a computer can carry.
- **Calls with audio at the same time** - calls with G.711 (PCMU, 20 ms packets, 50 packets a second) both ways, from `pcap/test.pcap`. The number of calls is raised step by step; in a step all the calls are up together for a minute. Each call is two RTP streams, one each way, so 100 calls are 100 streams sent and 100 received by each computer. A stream is 80 kbit/s at the IP level (200 bytes 50 times a second), about 95 kbit/s on Ethernet: 100 calls are about 9.5 Mbit/s each way, far below what a 1 Gbit/s network carries.

One computer makes the calls and the other answers; then they change roles. The weaker computer limits each run, so the CPU load of both is sampled every two seconds: it shows which one ran out.

The limits a step must keep

Limit	Meaning
asr>=99.5	Answer-seizure ratio: of the calls attempted, at least 99.5 % were answered (99 % in the audio test).
p95<=500	95 % of the SIP requests got their response within 500 ms.
loss<=0.5	At most 0.5 % of the RTP packets that should have come did not arrive.
mos>=4.0	The estimated voice quality (MOS, 1 to 4.5, calculated from loss and jitter) is at least 4.0. G.711 without loss is 4.4.

A step passes when the runs on both computers keep their limits: the caller checks the answers and the audio it receives, the answerer the audio it receives. The limit reported is the highest step that passed, narrowed down between it and the first that failed.

Measuring your own computers

1. On both computers start SIP Inspector as a service, with the address the other computer reaches it at:
`SIPInspDlxCLI -api 8080 -apiHost 0.0.0.0 -apiToken <token> -lip <address> -lp 5060 -lrtp 10000 -np`. The program folder needs only `pcap/test.pcap` and the licence; the script sends the scenarios.
2. Where Python 3 runs (either computer or a third one): `test/capacity.py --a http://<A>:8080 --b http://<B>:8080 --token <token> --a-name Laptop --b-name Server --repeat 3`.
3. The results are in `capacity_<date>/: summary.md` has the tables, `steps.csv` every step with its numbers and CPU load.

Note: Close other programs on both computers, use a wired network, and open the firewall for the API port, the SIP port and the RTP ports (from `-lrtp` up). Give Java enough memory for many calls (`-Xmx2g`).