
Tsung Documentation

Release 1.7.0

Nicolas Niclausse

Oct 08, 2017

Contents

1	Introduction	3
1.1	What is Tsung?	3
1.2	What is Erlang and why is it important for Tsung?	3
1.3	Tsung background	3
2	Features	5
2.1	Tsung main features	5
2.2	HTTP related features	6
2.3	WEBDAV related features	6
2.4	Jabber/XMPP related features	6
2.5	PostgreSQL related features	7
2.6	MySQL related features	7
2.7	Websocket related features	7
2.8	AMQP related features	7
2.9	MQTT related features	7
2.10	LDAP related features	8
2.11	Raw plugin related features	8
2.12	Complete reports set	8
2.13	Highlights	8
3	Installation	9
3.1	Dependencies	9
3.2	Compilation	10
3.3	Configuration	10
3.4	Running	10
3.5	Feedback	11
4	Benchmark Approach	13
4.1	HTTP/WebDAV	13
4.2	LDAP	14
4.3	PostgreSQL	14
4.4	MySQL	14
4.5	Jabber/XMPP	14
5	Using the proxy recorder	19
5.1	PostgreSQL	19
5.2	HTTP and WEBDAV	20

6	Understanding tsung.xml configuration file	21
6.1	File structure	21
6.2	Clients and server	22
6.3	Monitoring	25
6.4	Defining the load progression	26
6.5	Setting options	28
6.6	Sessions	32
6.7	Advanced Features	52
7	Statistics and Reports	63
7.1	File format	63
7.2	Available stats	64
7.3	Design	65
7.4	Generating the report	65
7.5	Tsung summary	66
7.6	Graphical overview	66
7.7	Tsung Plotter	69
7.8	RRD	69
8	References	71
9	Acknowledgments	73
10	Frequently Asked Questions	75
10.1	Can't start distributed clients: timeout error	75
10.2	Tsung crashes when I start it	76
10.3	Why do i have error_connect_emfile errors?	76
10.4	Tsung still crashes/fails when I start it!	77
10.5	Can I dynamically follow redirect with HTTP?	77
10.6	What is the format of the stats file tsung.log?	78
10.7	How can I compute percentile/quartiles/median for transactions or requests response time?	78
10.8	How can I specify the number of concurrent users?	79
10.9	SNMP monitoring doesn't work?!	79
10.10	How can i simulate a fix number of users?	80
11	Errors list	81
11.1	error_closed	81
11.2	error_inet_<ERRORNAME>	81
11.3	error_unknown_data	81
11.4	error_unknown_msg	81
11.5	error_unknown	81
11.6	error_repeat_<REPEATNAME>	81
11.7	error_send_<ERRORNAME>	82
11.8	error_send	82
11.9	error_connect_<ERRORNAME>	82
11.10	error_no_online	82
11.11	error_no_offline	82
11.12	error_no_free_userid	82
11.13	error_next_session	82
11.14	error_mysql_<ERRNO>	82
11.15	error_mysql_badpacket	82
11.16	error_pgsq	83
12	Changelog	85

13	tsung-1.0.dtd	87
14	Indices and tables	97

Everything you need to know about Tsung

About Tsung

Tsung is a high-performance benchmark framework for various protocols including HTTP, XMPP, LDAP, etc

- **Website:** tsung.erlang-projects.org
- **Source code:** github.com/processone/tsung
- **Bugtracker:** ‘<<https://github.com/processone/tsung/issues>>’

What is Tsung?

Tsung (formerly IDX-Tsunami) is a distributed load testing tool. It is protocol-independent and can currently be used to stress HTTP, WebDAV, SOAP, PostgreSQL, MySQL, AMQP, MQTT, LDAP and Jabber/XMPP servers.

It is distributed under the GNU General Public License version 2.

What is Erlang and why is it important for Tsung?

Tsung's main strength is its ability to simulate a huge number of simultaneous user from a single machine; moreover, you can distribute the users on cluster for machines. When used on cluster, you can generate a really impressive load on a server with a modest cluster, easy to set-up and to maintain. You can also use Tsung on a cloud like EC2.

Tsung is developed in Erlang and this is where the power of Tsung resides.

Erlang is a *concurrency-oriented* programming language. Tsung is based on the Erlang OTP (Open Telecom Platform) and inherits several characteristics from Erlang:

Performance Erlang has been made to support hundred thousands of lightweight processes in a single virtual machine.

Scalability Erlang runtime environment is naturally distributed, promoting the idea of process's location transparency.

Fault-tolerance Erlang has been built to develop robust, fault-tolerant systems. As such, wrong answer sent from the server to Tsung does not make the whole running benchmark crash.

More information on Erlang on <http://www.erlang.org>.

Tsung background

History:

- Tsung development was started by Nicolas Niclausse in 2001 as a distributed jabber load stress tool for internal use at <http://IDEALX.com/> (now OpenTrust). It has evolved as an open-source multi-protocol load testing tool several months later. The HTTP support was added in 2003, and this tool has been used for several industrial projects. It is now hosted on github, and several companies provide professional support. The list of contributors is available in the source archive at <https://github.com/processone/tsung/blob/master/CONTRIBUTORS>.
- It is an industrial strength implementation of a *stochastic model* for real users simulation. User events distribution is based on a Poisson Process. More information on this topic in:

Z. Liu, N. Niclausse, and C. Jalpa-Villanueva. **Traffic Model and Performance Evaluation of Web Servers.** *Performance Evaluation, Volume 46, Issue 2-3, October 2001.*
- This model has already been tested in the INRIA WAGON research prototype (Web trAffic GeneratOr and beNchmark). WAGON was used in the <http://www.vthd.org/> project (Very High Broadband IP/WDM test platform for new generation Internet applications, 2000-2004).

Tsung has been used for very high load tests:

- *Jabber/XMPP* protocol:
 - 90,000 simultaneous Jabber users on a 4-node Tsung cluster (3xSun V240 + 1 Sun V440).
 - 10,000 simultaneous users. Tsung was running on a 3-computers cluster (CPU 800MHz).
 - 2,000,000 concurrent users on a single m4.10xlarge instance on EC2 to test ejabberd scalability
- *HTTP and HTTPS* protocol:
 - 12,000 simultaneous users. Tsung were running on a 4-computers cluster (in 2003). The tested platform reached 3,000 https requests per second.
 - 10 million simultaneous users running on a 75-computers cluster, generating more than one million requests per second.

Tsung has been used at:

- *DGI* (Direction Générale des impôts): French finance ministry
- *Cap Gemini Ernst & Young*
- *IFP* (Institut Français du Pétrole): French Research Organization for Petroleum
- *LibertySurf*
- *Sun* (TM) for their Moodlerooms platform on Niagara processors: <https://blogs.oracle.com/kevinr/resource/Moodle-Sun-RA.pdf>
- and many other companies

Tsung main features

- *High Performance*: Tsung can simulate a huge number of simultaneous users per physical computer: It can simulate thousands of users on a single CPU (Note: a simulated user is not always active: it can be idle during a `thinktime` period). Traditional injection tools can hardly go further than a few hundreds (Hint: if all you want to do is requesting a single URL in a loop, use **ab**; but if you want to build complex scenarios with extended reports, Tsung is for you).
- *Distributed*: the load can be distributed on a cluster of client machines
- *Multi-Protocols* using a plug-in system: HTTP (both standard web traffic and SOAP), WebDAV, Jabber/XMPP and PostgreSQL are currently supported. LDAP and MySQL plugins were first included in the 1.3.0 release.
- *SSL support*
- *Several IP addresses* can be used on a single machine using the underlying OS IP Aliasing
- *OS monitoring* (CPU, memory and network traffic) using Erlang agents on remote servers or *SNMP*
- *XML configuration system*: complex user's scenarios are written in XML. Scenarios can be written with a simple browser using the Tsung recorder (HTTP and PostgreSQL only).
- *Dynamic scenarios*: You can get dynamic data from the server under load (without writing any code) and re-inject it in subsequent requests. You can also loop, restart or stop a session when a string (or regexp) matches the server response.
- *Mixed behaviours*: several *sessions* can be used to simulate different type of users during the same benchmark. You can define the proportion of the various behaviours in the benchmark scenario.
- *Stochastic processes*: in order to generate a realistic traffic, user thinktimes and the arrival rate can be randomized using a probability distribution (currently exponential)

HTTP related features

- HTTP/1.0 and HTTP/1.1 support
- GET, POST, PUT, DELETE, HEAD, OPTIONS and PATCH requests
- Cookies: Automatic cookies management (but you can also manually add more cookies)
- ‘GET If-modified since’ type of request
- WWW-authentication Basic and Digest. OAuth 1.0
- User Agent support
- Any HTTP Headers can be added
- Proxy mode to record sessions using a Web browser
- SOAP support using the HTTP mode (the SOAPAction HTTP header is handled).
- HTTP server or proxy server load testing.

WEBDAV related features

The WebDAV ([RFC 4918](#)) plugin is a superset of the HTTP plugin. It adds the following features (some versioning extensions to WebDAV ([RFC 3253](#)) are also supported):

- Methods implemented: DELETE, CONNECT, PROPFIND, PROPPATCH, COPY, MOVE, LOCK, UNLOCK, MKCOL, REPORT, OPTIONS, MKACTIVITY, CHECKOUT, MERGE
- Recording of DEPTH, IF, TIMEOUT OVERWRITE, DESTINATION, URL and LOCK-TOKEN Headers.

Jabber/XMPP related features

- Authentication (plain-text, digest and sip-digest). STARTTLS
- presence and register messages
- Chat messages to online or offline users
- MUC: join room, send message in room, change nickname
- Roster set and get requests
- Global users’ synchronization can be set on specific actions
- BOSH & XMPP over Websocket
- raw XML messages
- PubSub
- Multiple vhost instances supported
- privacy lists: get all privacy list names, set list as active

PostgreSQL related features

- Basic and MD5 Authentication
- Simple Protocol
- Extended Protocol (new in version **1.4.0**)
- Proxy mode to record sessions

MySQL related features

This plugin is experimental. It works only with MySQL version 4.1 and higher.

- Secured Authentication method only (MySQL >= 4.1)
- Basic Queries

Websocket related features

This plugin is experimental. It only supports [RFC 6455](#) currently. For used as a server type, it works like other transport protocol like tcp and udp, any application specific protocol data can be send on it.

You can find examples used as session type in examples/websocket.xml.

- Both as a server type and session type

AMQP related features

This plugin is experimental. It only supports AMQP-0.9.1 currently. You can find examples in examples/amqp.xml.

- Basic publish and consume
- Publisher confirm and consumer ack
- QoS

MQTT related features

This plugin is experimental. It supports MQTT V3.1. You can find examples in examples/mqtt.xml.

- Connect to mqtt broker with options
- Publish mqtt messages to the broker
- Subscribe/unsubscribe topics
- Support QoS 0 and QoS 1

LDAP related features

- Bind
- Add, modify and search queries
- Starttls

Raw plugin related features

- TCP / UDP / SSL compatible
- raw messages
- no_ack, local or global ack for messages

Complete reports set

Measures and statistics produced by Tsung are extremely feature-full. They are all represented as a graphic. Tsung produces statistics regarding:

- *Performance*: response time, connection time, decomposition of the user scenario based on request grouping instruction (called *transactions*), requests per second
- *Errors*: Statistics on page return code to trace errors
- *Target server behaviour*: An Erlang agent can gather information from the target server(s). Tsung produces graphs for CPU and memory consumption and network traffic. SNMP and munin is also supported to monitor remote servers.

par Note that Tsung takes care of the synchronization process by itself. Gathered statistics are «synchronized».

It is possible to generate graphs during the benchmark as statistics are gathered in real-time.

Highlights

Tsung has several advantages over other injection tools:

- *High performance and distributed benchmark*: You can use Tsung to simulate tens of thousands of virtual users.
- *Ease of use*: The hard work is already done for all supported protocol. No need to write complex scripts. Dynamic scenarios only requires small trivial piece of code.
- *Multi-protocol support*: Tsung is for example one of the only tool to benchmark SOAP applications
- *Monitoring* of the target server(s) to analyze the behaviour and find bottlenecks. For example, it has been used to analyze cluster symmetry (is the load properly balanced ?) and to determine the best combination of machines on the three cluster tiers (Web engine, EJB engine and database)

This package has been tested on Linux, FreeBSD and Solaris. A port is available on Mac OS X. It should work on Erlang supported platforms (Linux, Solaris, *BSD, Win32 and Mac OS X).

On Mac OS X you can install Tsung via Homebrew (<http://brew.sh/>): **brew install tsung**.

Dependencies

- **Erlang/OTP R16B03** and up ([download](#)).
- **pgsql module** made by Christian Sunesson (for the PostgreSQL plugin): sources available at <http://jungerl.sourceforge.net/>. The module is included in the source and binary distribution of Tsung. It is released under the EPL License.
- **mysql module** made by Magnus Ahlertorp & Fredrik Thulin (for the mysql plugin): sources available at <http://www.stacken.kth.se/projekt/yxa/>. The modified module is included in the source and binary distribution of Tsung. It is released under the three-clause BSD License.
- **mochiweb** libs (for XPath parsing, optionally used for dynamic variables in the HTTP plugin): sources available at <https://github.com/mochi/mochiweb>. The module is included in the source and binary distribution of Tsung. It is released under the MIT License.
- **gnuplot** and **perl5** (optional; for graphical output with `tsung_stats.pl` script). The Template Toolkit is used for HTML reports (see <http://template-toolkit.org/>).
- **python** and **matplotlib** (optional; for graphical output with `tsung-plotter`).
- for distributed tests, you need SSH access to remote machines without password (use a RSA/DSA key without passphrase or `ssh-agent`). Alternatively `rsh` is also supported.
- `bash`

Compilation

To compile Tsung, just download the latest version from <http://tsung.erlang-projects.org/dist/> and run:

```
./configure
make
make install
```

If you want to download the latest development version, use git:

git clone <https://github.com/processone/tsung.git>, see also <https://github.com/processone/tsung>.

You can also build packages with **make deb** (on Debian and Ubuntu) and **make rpm** (on Fedora, RHEL and other rpm based distribution).

Configuration

The default configuration file is `~/.tsung/tsung.xml` (there are several sample files in `/usr/share/doc/tsung/examples`).

Log files are saved in `~/.tsung/log/`. A new subdirectory is created for each test using the current date and time as name, e.g. `~/.tsung/log/20040217-0940`.

Running

Two commands are installed in the directory `$PREFIX/bin`: `tsung` and `tsung-recorder`. A man page is available for both commands.

```
$ tsung -h
Usage: tsung <options> start|stop|debug|status
Options:
  -f <file>      set configuration file (default is ~/.tsung/tsung.xml)
                  (use - for standard input)
  -l <logdir>     set log directory where YYYYMMDD-HHMM dirs are created (default is ~
→ ~/.tsung/log/)
  -i <id>        set controller id (default is empty)
  -r <command>    set remote connector (default is ssh)
  -s            enable erlang smp on client nodes
  -p <max>       set maximum erlang processes per vm (default is 250000)
  -X <dir>       add additional erlang load paths (multiple -X arguments allowed)
  -m <file>      write monitoring output on this file (default is tsung.log)
                  (use - for standard output)
  -F            use long names (FQDN) for erlang nodes
  -L <lifetime>  SSL session lifetime (600sec by default)
  -w <delay>     warmup delay (default is 1 sec)
  -n            disable web GUI (started by default on port 8091)
  -k            keep web GUI (and controller) alive after the test has finished
  -v            print version information and exit
  -6            use IPv6 for Tsung internal communications
  -x <tags>      list of requests tag to be excluded from the run (separated by ↵
→ comma)
  -t <min>       erlang inet listening TCP port min (default: 64000)
```


<pre>-T <max> erlang inet listening TCP port max (default: 65500) -h display this help and exit</pre>

A typical way of using tsung is to run: **tsung -f myconfigfile.xml start**.

The command will print the current log directory created for the test, and wait until the test is over. By default an embedded web server will be started on the controller node and will listen on the 8091 port (this can be disabled with the *-n* option).

Feedback

Use the [Tsung mailing list](https://github.com/processone/tsung/issues) if you have suggestions or questions about Tsung. You can also use the bug tracker available at <https://github.com/processone/tsung/issues>

You can also try the #tsung IRC channel on Freenode.

HTTP/WebDAV

Benchmarking a Web server

1. Record one or more sessions: start the recorder with: **tsung-recorder start**, and then configure your browser to use Tsung proxy recorder (the listen port is 8090). A session file will be created. For HTTPS recording, use `http://` instead of `https://` in your browser.
2. Edit / organize scenario, by adding recorded sessions in the configuration file.
3. Write small code for dynamic parts if needed and place dynamic mark-up in the scenario.
4. Test and adjust scenario to have a nice progression of the load. This is highly dependent of the application and of the size of the target server(s). Calculate the normal duration of the scenario and use the interarrival time between users and the duration of the phase to estimate the number of simultaneous users for each given phase.
5. Launch benchmark with your first application parameters setup: **tsung start** (run **man tsung** for more options).
6. Wait for the end of the test or stop by hand with **tsung stop** (reports can also be generated during the test (see *Statistics and Reports*): the statistics are updated every 10 seconds). For a brief summary of the current activity, use **tsung status**.
7. Analyze results, change parameters and relaunch another benchmark.

WebDAV

It's the same approach as HTTP: first you start to record one or more sessions with the *recorder*: **tsung-recorder -p webdav start**.

Benchmarking a proxy server

By default, the HTTP plugin is used to benchmark HTTP servers. But you can also benchmark HTTP Proxy servers. To do that, you must add in the `options` section:

```
<option type="ts_http" name="http_use_server_as_proxy" value="true"></option>
```

LDAP

An LDAP plugin for the recorder is not yet implemented, so you have to write the session by yourself; see section [Authentication](#) for more information.

PostgreSQL

It's the same approach as HTTP: first you start to record one or more sessions with the recorder: **tsung-recorder -p pgsql start**.

This will start a proxy listening to port 8090 and will proxy requests to `127.0.0.0:5432`.

To choose another port and/or address: **tsung-recorder -L 5432 -I 10.6.1.1 -P 5433 -p pgsql start**.

This will start a proxy listening to port 5432 and will proxy requests to `10.6.1.1:5433`.

MySQL

A MySQL plugin for the recorder is not yet implemented, so you have to write the session by yourself; see section [MySQL](#) for more information.

Jabber/XMPP

Overview

This paragraph explains how to write a session for Jabber/XMPP.

There are two differences between HTTP and Jabber testing:

- There is no recorder for Jabber, so you have to write your sessions by hand. An example is provided in [Jabber/XMPP](#).
- The Jabber plugin does not parse XML; instead it uses packet acknowledgments.

Acknowledgments of messages

Since the Jabber plugin does not parse XML (historically, it was for performance reasons), you must have a way to tell when a request is finished. There are 3 possibilities using the `ack` attribute:

- `ack="local"` as soon as a packet is received from the server, the request is considered as completed. Hence if you use a local ack with a request that do not require a response from the server (presence for ex.), it will wait forever (or until a timeout is reached).
- `ack="no_ack"` as soon as the request is send, it is considered as completed (do not wait for incoming data).
- `ack="global"` synchronized users. its main use is for waiting for all users to connect before sending messages. To do that, set a request with global ack (it can be the first presence msg:

```
<request> <jabber type="presence" ack="global"/> </request>
```

You also have to specify the number of users to be connected:

```
<option type="ts_jabber" name="global_number" value="100"></option>
```

To be sure that exactly `global_number` users are started, add the `maxnumber` attribute to users:

```
<users maxnumber="100" interarrival="1.0" unit="second"></users>
```

If you do not specify `maxnumber`, the global ack will be reset every `global_number` users.

Bidirectional Presence

New in 1.2.2: This version adds an new option for a session. if you set the attribute `bidi` (for bidirectional) in the `<session>` tag: `<session ... bidi="true">`, then incoming messages from the server will be analyzed. Currently, only roster subscription requests are handled: if a user received a subscription request (`<presence ... type="subscribe">`), it will respond with a `<presence ... type="subscribed">` message.

Status: Offline, Connected and Online

You can send messages to offline or online users. A user is considered online when he has send a `presence:initial` message (before this message , the state of the user is `connected`).

If you want to switch back to **connected** before going **offline**, you can use a **presence:final** message:

presence:final does two things:

- It removes the client from the list of Online users, and moves them into the list of Connected users.
- It sends a broadcast presence update of `type="unavailable"`.

presence:final is optional.

Warning: this is new in 1.2.0, in earlier version, only 2 status were available: online and offline; a user was considered online as soon as it was connected.

Authentication

Below are configuration examples for the possible authentication methods. Note: the regular expressions used here are only examples - they may need to be altered depending on how a particular server implementation composes messages (see also [Websocket options](#) for password settings).

- **plain authentication** - sends clear-text passwords:

```

<session probability="100" name="jabber-plain" type="ts_jabber">

  <request> <jabber type="connect" ack="local"></jabber> </request>

  <thinktime value="2"></thinktime>

  <transaction name="auth_plain">
    <request> <jabber type="auth_get" ack="local"></jabber> </request>
    <request> <jabber type="auth_set_plain" ack="local"></jabber> </request>
  </transaction>
  ...
</session>

```

- **digest authentication** as described in XMPP JEP-0078: Non-SASL Authentication <http://www.jabber.org/jeps/jep-0078.html>

```

<session probability="100" name="jabber-digest" type="ts_jabber">

  <!-- regexp captures stream ID returned by server -->
  <request>
    <dyn_variable name="sid" re="&lt;stream:stream id=&quot;(.*)&quot;;&lt;br />
    <!-- xmlns:stream -->
    <jabber type="connect" ack="local"></jabber>
  </request>

  <thinktime value="2"></thinktime>

  <transaction name="auth_digest">
    <request> <jabber type="auth_get" ack="local"></jabber> </request>
    <request subst="true"> <jabber type="auth_set_digest" ack="local"></jabber> </
    <!-- request -->
  </transaction>
  ...
</session>

```

- **sip-digest authentication**

```

<session probability="100" name="jabber-sipdigest" type="ts_jabber">

  <request> <jabber type="connect" ack="local"></jabber> </request>

  <thinktime value="2"></thinktime>

  <transaction name="auth_sipdigest">
    <!-- regexp captures nonce value returned by server -->
    <request>
      <dyn_variable name="nonce"
        re="&lt;Nonce encoding=&quot;hex&quot;&gt;(.*)&lt;/Nonce&gt;"/>
      <jabber type="auth_get" ack="local"></jabber>
    </request>
    <request subst="true"> <jabber type="auth_set_sip" ack="local"></jabber> </
    <!-- request -->
  </transaction>
  ...
</session>

```

Privacy list testing

There are two actions available to allow for rudimentary privacy lists load testing:

- **privacy:get_names** gets the list of all names .. of privacy lists stored by the server for a given user
- **privacy:set_active** sets a list with a predefined name as active. The list name is determined from the JID, e.g. if the user's JID is "john@average.com" then the list name is "john@average.com_list". One should take care of properly seeding the server database in order to ensure that such a list exists.

CHAPTER 5

Using the proxy recorder

The recorder has three plugins: for HTTP, WebDAV and for PostgreSQL.

To start it, run **tsung-recorder -p <PLUGIN> start**, where **PLUGIN** can be *http*, *webdav* or *pgsql* for PostgreSQL. The default plugin is **http**.

The proxy is listening to port **8090**. You can change the port with **-L portnumber**.

To stop it, use **tsung-recorder stop**.

The recorded session is created as `~/.tsung/tsung_recorderYYYYMMDD-HH:MM.xml`; if it doesn't work, take a look at `~/.tsung/log/tsung.log-tsung_recorder@hostname`

During the recording, you can add custom tag in the XML file, this can be useful to set transactions or comments:
tsung-recorder record_tag "<transaction name='login'>"

Once a session has been created, you can insert it in your main configuration file, either by editing by hand the file, or by using an ENTITY declaration, like:

```
<!DOCTYPE tsung SYSTEM "/usr/share/tsung/tsung-1.0.dtd" [  
  <!ENTITY mysession1 SYSTEM "/home/nniclausse/.tsung/tsung_recorder20051217-13:11.xml"  
  ↪ ">  
>  
...  
<sessions>  
  &mysession1;  
</sessions>
```

PostgreSQL

For PostgreSQL, the proxy will connect to the server at IP 127.0.0.1 and port 5432. Use **-I serverIP** to change the IP and **-P portnumber** to change the port.

HTTP and WEBDAV

For HTTPS recording, use **http://** instead of **https://** in your browser

New in 1.2.2: For HTTP, you can configure the recorder to use a parent proxy (but this will not work for https). Add the `-u` option to enable parent proxy, and use **-I serverIP** to set the IP and **-P portnumber** to set the port of the parent.

Understanding tsung.xml configuration file

File structure

The default encoding is utf-8. You can use a different encoding, like in:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

Scenarios are enclosed into **tsung** tags:

```
<?xml version="1.0"?>
<!DOCTYPE tsung SYSTEM "/usr/share/tsung/tsung-1.0.dtd" [] >
<tsung loglevel="info">
...
</tsung>
```

If you add the attribute **dumptraffic="true"**, all the traffic will be logged to a file.

Warning: this will considerably slow down Tsung, so use with care. It is useful for debugging purpose. You can use the attribute **dumptraffic="light"** to dump only the first 44 bytes.

Since version **1.4.0**, you have also a specific logging per protocol, using **dumptraffic="protocol"**. It's currently only implemented for HTTP: this will log all requests in a CSV file, with the following data:

```
#date;pid;id;http method;host;URL;HTTP status;size;duration;transaction;match;error;
↪tag
```

Where:

field	description
date	timestamp at the end of the request (seconds since 1970-01-01 00:00:00 UTC)
pid	erlang process id
id	tsung user id
host	server hostname
url	URL (relative)
HTTP	status HTTP reponse status (200, 304, etc.)
size	reponse size (in bytes)
duration	request duration (msec)
transaction	name of the transaction (if any) this request was made in
match	if a match is defined in the request: match nomatch (last <match> if several are defined)
error	name of http error (or empty)
tag	tag name if the request was tagged; empty otherwise

Warning: In the general case (several Tsung clients used), the resulting file will not be sorted, so you may have to sort it before analyzing it.

For heavy load testing (tens of thousands requests per second), the **protocol** logging may overload the controller. In this case, you can use **protocol_local** instead. In this case, the log files will be written on each slave locally. You will have to manually merged the logs at the end of the test.

The **loglevel** can also have a great impact on performance: For high load, **warning** is recommended.

Possible values are:

- emergency
- critical
- error
- warning
- notice (*default*)
- info
- debug

For REALLY verbose logging, recompile tsung with **make debug** and set **loglevel** to **debug**.

Clients and server

Scenarios start with clients (Tsung cluster) and server definitions:

Basic setup

For non distributed load, you can use a basic setup like:

```
<clients>
  <client host="localhost" use_controller_vm="true"/>
</clients>

<servers>
```

```
<server host="192.168.1.1" port="80" type="tcp"></server>
</servers>
```

This will start the load on the same host and on the same Erlang virtual machine as the controller.

The server is the entry point into the cluster. You can add several servers, by default each server will have a weight of 1, and each session will choose a server randomly according to the weight. You can set a weight for each server like this (weight can be an integer or a float):

```
<servers>
  <server host="server1" port="80" type="tcp" weight="4"></server>
  <server host="server2" port="80" type="tcp" weight="1"></server>
</servers>
```

(in version older than **1.5.0**, the `weight` option was not implemented and a round robin algorithm was used to choose the server).

Type can be `tcp`, `ssl`, `udp` (for IPv6, use `tcp6`, `ssl6` or `udp6` ; only available in version **1.4.2** and newer) or `websocket` (only available in version **1.5.0** and newer))

There's also a specific type for BOSH: `bosh` for unencrypted BOSH, and `bosh_ssl` for encrypted connection

Advanced setup

The next example is more complex, and use several features for advanced distributed testing:

```
<clients>
  <client host="louxor" weight="1" maxusers="800">
    <ip value="10.9.195.12"></ip>
    <ip value="10.9.195.13"></ip>
  </client>
  <client host="memphis" weight="3" maxusers="600" cpu="2"/>
</clients>

<servers>
  <server host="10.9.195.1" port="8080" type="tcp"></server>
</servers>
```

Several virtual IP can be used to simulate more machines. This is very useful when a load-balancer use the client's IP to distribute the traffic among a cluster of servers. **New in 1.1.1:** IP is no longer mandatory. If not specified, the default IP will be used.

New in 1.4.0: You can use `<ip scan="true" value="eth0"/>` to scan for all the IP aliases on a given interface (`eth0` in this example).

In this example, a second machine is used in the Tsung cluster, with a higher weight, and 2 cpus. Two Erlang virtual machines will be used to take advantage of the number of CPU.

Note: Even if an Erlang VM is now able to handle several CPUs (erlang SMP), benchmarks shows that it's more efficient to use one VM per CPU (with SMP disabled) for tsung clients. Only the controller node is using SMP erlang. Therefore, `cpu` should be equal to the number of cores of your nodes. If you prefer to use erlang SMP, add the `-s` option when starting tsung (and don't set `cpu` in the config file).

By default, the load is distributed uniformly on all CPU (one CPU per client by default). The weight parameter (integer) can be used to take into account the speed of the client machine. For instance, if one real client has a weight of 1 and the other client has a weight of 2, the second one will start twice the number of users as the first (the

proportions will be 1/3 and 2/3). In the earlier example where for the second client has 2 CPU and weight=3, the weight is equal to 1.5 for each CPU.

direct ip

Sometimes, it can be a problem to use hostnames for all tsung clients (if you don't have a DNS, you must edit /etc/hosts on all nodes). Since **version in 1.7.0**, you can use direct IP instead of hostnames.

To do this, you should use the `-I` parameter when starting Tsung,

```
tsung -I Your_Server_IP -f tsung.xml start
```

eg:

```
tsung -I 10.10.10.10 -f tsung.xml start
```

You controller node name is therefore: `tsung_controller@10.10.10.10`. For clients, put the IP like this:

```
<clients>
  <client host="10.10.10.11" maxusers="120000" cpu="7" weight="4"/>
  <client host="10.10.10.12" maxusers="120000" cpu="7" weight="4"/>
</clients>
```

iprange

If you have many IPs (thousands of IPs), the `ip scan` option can be slow ; in this case you can use the `iprange` tag to generate a random IP in a given range:

```
<iprange version="v4" value="172.28.1-20.0-254"/>
```

In the given example, the third and last part of the IPv4 address will be random in the given range.

maxusers

The `maxusers` parameter is used to bypass the limit of maximum number of sockets opened by a single process (1024 by default on many OS) and the lack of scalability of the `select` system call. When the number of users is higher than the limit, a new erlang virtual machine will be started to handle new users. The default value of `maxusers` is 800. Nowadays, with kernel polling enable, you can and should use a very large value for `maxusers` (30000 for example) without performance penalty (but don't forget to raise the limit of the OS with `ulimit -n`, see also *Why do i have error_connect_emfile errors?*).

Note: If you are using a tsung master with slaves, the master distributes sessions to slaves. If a session contains multiples requests, a slave will execute each of these requests in order.

Running Tsung with a job scheduler

Tsung is able to get its client node list from a batch/job scheduler. It currently handle PBS/torque, LSF and OAR. To do this, set the `type` attribute to `batch`, e.g.:

```
<client type="batch" batch="torque" maxusers="30000">
```

If you need to scan IP aliases on nodes given by the batch scheduler, use `scan_intf` like this:

```
<client type="batch" batch="torque" scan_intf='eth0' maxusers="30000">
```

Monitoring

Tsung is able to monitor remote servers using several backends that communicates with remote agent. This is configured in the `<monitoring>` section. Available statistics are: CPU activity, load average and memory usage.

Note that you can get the nodes to monitor from a job scheduler, like:

```
<monitor batch="true" host="torque" type="erlang"></monitor>
```

Several types of remote agents are supported (erlang is the default):

Erlang

The remote agent is started by Tsung. It use erlang communications to retrieve statistics of activity on the server. For example, here is a cluster monitoring definition based on Erlang agents, for a cluster of 6 computers:

```
<monitoring>
  <monitor host="geronimo" type="erlang"></monitor>
  <monitor host="bigfoot-1" type="erlang"></monitor>
  <monitor host="bigfoot-2" type="erlang"></monitor>
  <monitor host="f14-1" type="erlang"></monitor>
  <monitor host="f14-2" type="erlang"></monitor>
  <monitor host="db" type="erlang"></monitor>
</monitoring>
```

Note: monitored computers needs to be accessible through the network, and erlang communications must be allowed (no firewall is better). SSH (or rsh) needs to be configured to allow connection without password on. **You must use the same version of Erlang/OTP on all nodes otherwise it may not work properly!**

If you can't have erlang installed on remote servers, you can use one of the other available agents.

New in version 1.5.1.

erlang monitoring includes now an option to monitor a mysql db with mysqladmin. Use it like this:

```
<monitor host="db" type="erlang"></monitor>
  <mysqladmin port="3306" username="root" password="sesame" />
</monitor>
```

Availabe stats: number of mysql threads and Questions (queries)

SNMP

The `type` keyword `snmp` can replace the `erlang` keyword, if SNMP monitoring is preferred. They can be mixed. **Since version 1.2.2**, you can customize the SNMP version, community and port number. It uses the Management Information Base (MIB) provided in `net-snmp` (see also *SNMP monitoring doesn't work?!*).

```
<monitoring>
  <monitor host="geronimo" type="snmp"/>
  <monitor host="f14-2" type="erlang"></monitor>
  <monitor host="db" type="snmp">
    <snmp version="v2" community="mycommunity" port="11161"/>
  </monitor>
</monitoring>
```

The default version is v1, default community public and default port 161.

Since version **1.4.2**, you can also customize the object identifiers (OID) retrieved from the SNMP server, using one or several `oid` element:

```
<monitor host="127.0.0.1" type="snmp">
  <snmp version="v2">
    <oid value="1.3.6.1.4.1.42.2.145.3.163.1.1.2.11.0"
        name="heapused" type="sample" eval="fun(X)-> X/100 end."/>
  </snmp>
</monitor>
```

type can be sample, counter or sum, and optionally you can define a function (with erlang syntax) to be applied to the value (eval attribute).

Munin

New in version 1.3.1.

Tsung is able to retrieve data from a munin-node agent (see <http://munin-monitoring.org/wiki/munin-node>). The type keyword must be set to munin, for example:

```
<monitoring>
  <monitor host="geronimo" type="munin"/>
  <monitor host="f14-2" type="erlang"></monitor>
</monitoring>
```

Defining the load progression

Randomly generated users

The load progression is set-up by defining several arrival phases:

```
<load>
  <arrivalphase phase="1" duration="10" unit="minute">
    <users interarrival="2" unit="second"></users>
  </arrivalphase>

  <arrivalphase phase="2" duration="10" unit="minute">
    <users interarrival="1" unit="second"></users>
  </arrivalphase>

  <arrivalphase phase="3" duration="10" unit="minute">
    <users interarrival="0.1" unit="second"></users>
  </arrivalphase>
</load>
```


With this setup, during the first 10 minutes of the test, a new user will be created every 2 seconds, then during the next 10 minutes, a new user will be created every second, and for the last 10 minutes, 10 users will be generated every second. The test will finish when all users have ended their session.

You can also use `arrivalrate` instead of `interarrival`. For example, if you want 10 new users per second, use:

```
<arrivalphase phase="1" duration="10" unit="minute">
  <users arrivalrate="10" unit="second"></users>
</arrivalphase>
```

You can limit the number of users started for each phase by using the `maxnumber` attribute, just like this:

```
<arrivalphase phase="1" duration="10" unit="minute">
  <users maxnumber="100" arrivalrate="10" unit="second"></users>
</arrivalphase>
<arrivalphase phase="2" duration="10" unit="minute">
  <users maxnumber="200" arrivalrate="10" unit="second"></users>
</arrivalphase>
```

In this case, only 100 users will be created in the first phases, and 200 more during the second phase.

The complete sequence can be executed several times using the `loop` attribute in the `load` tag (`loop='2'` means the sequence will be looped twice, so the complete load will be executed 3 times) (feature available since version 1.2.2).

The load generated in terms of HTTP requests / seconds will also depend on the mean number of requests within a session (if you have a mean value of 100 requests per session and 10 new users per seconds, the theoretical average throughput will be 1000 requests/ sec).

New in version 1.5.1.

You can also override the probability settings of sessions within a specific phase, using `session_setup`:

```
<arrivalphase phase="3" duration="1" unit="minute">
  <session_setup name="http_test_1" probability="80"/>
  <session_setup name="fake" probability="20"/>
  <users interarrival="1" unit="second"/>
</arrivalphase>
```

New in version 1.7.0.

By default, a phase ends when its duration has been reached, even if all started sessions during the phase are not finished. You can override this behavior. If you want to start a new phase only after all generated users in the previous phase have finished their sessions, use the `wait_all_sessions_end` attribute, like this:

```
<arrivalphase phase="1" duration="10" unit="minute" wait_all_sessions_end="true">
  <users interarrival="1" unit="second"/>
</arrivalphase>
<arrivalphase phase="2" duration="10" unit="minute">
  <users interarrival="5" unit="second"/>
</arrivalphase>
```

(In this case, the real duration of the phase 1 will probably be higher than its configured one.)

Statically generated users

If you want to start a given session (see [Sessions](#)) at a given time during the test, it is possible since version 1.3.1:

```
<load>
  <arrivalphase phase="1" duration="10" unit="minute">
    <users interarrival="2" unit="second"></users>
  </arrivalphase>
  <user session="http-example" start_time="185" unit="second"></user>
  <user session="http-example" start_time="10" unit="minute"></user>
  <user session="foo" start_time="11" unit="minute"></user>
</load>
<sessions>
  <session name="http-example" probability="0" type="ts_http">
    <request> <http url="/" method="GET"></http> </request>
  </session>
  <session name="foobar" probability="0" type="ts_http">
    <request> <http url="/bar" method="GET"></http> </request>
  </session>
  <session name="foo" probability="100" type="ts_http">
    <request> <http url="/" method="GET"></http> </request>
  </session>
</sessions>
```

In this example, we have two sessions, one has a “0” probability (and therefore will not be used in the first phase), and the other 100%. We define 3 users starting respectively 3mn and 5 seconds after the beginning of the test (using the `http-example` session), one starting after 10 minutes, and a last one starting after 11 minutes (using the `foo` session this time)

New in version 1.5.1.

If you want to start several sessions at once, and if the name of these sessions starts with the same prefix, you can use a wildcard. Given the previous sessions, this example will start two users (one with `foo` session, and one with `foobar` session) at starttime +10s.

```
<user session="foo*" start_time="10" unit="second"/>
```

Duration of the load test

By default, tsung will end when all started users have finished their session. So it can be much longer than the duration of arrivalphases. If you want to stop Tsung after a given duration (even if phases are not finished or if some sessions are still actives), you can do this with the `duration` attribute in `load` (**feature added in 1.3.2**):

```
<load duration="1" unit="hour">
  <arrivalphase phase="1" duration="10" unit="minute">
    <users interarrival="2" unit="second"></users>
  </arrivalphase>
</load>
```

Currently, the maximum value for duration is a little bit less than 50 days. `unit` can be `second`, `minute` or `hour`.

Setting options

Thinktimes, SSL, Buffers

Default values can be set-up globally: `thinktime` between requests in the scenario, SSL cipher algorithms (use `ssl:cipher_suites(openssl)` to get a list of available ciphers), TCP/UDP buffer sizes (the default value is 32KB). These values overrides those set in session configuration tags if `override` is true.

```
<option name="thinktime" value="3" random="false" override="true"/>
<option name="ssl_ciphers"
      value="EXP1024-RSA,EDH-RSA-DES-CBC3-SHA"/>
<option name="tcp_snd_buffer" value="16384"/>
<option name="tcp_rcv_buffer" value="16384"/>
<option name="udp_snd_buffer" value="16384"/>
<option name="udp_rcv_buffer" value="16384"/>
```

New in version 1.6.0.

You can disable the SSL session cache (it is enabled by default)

```
<option name="ssl_reuse_sessions" value="false"/>
```

You can specify which SSL protocol you want use. Use `ssl:versions()`. to get a list of available ssl protocols.

```
<option name="ssl_versions" value="'tlsv1.2'"/>
```

You can also use the command line option `-L <value>` to change the session lifetime in the cache (10mn by default); value must be in seconds.

You can also change the way Tsung starts remote beams. By default, Tsung will start at most 20 ssh process per core of the controller. If you manage hundreds of clients, you may want to raise this value with `max_ssh_startup_per_core` (or decrease it if you wish)

```
<option name="max_ssh_startup_per_core" value="100"/>
```

Timeout for TCP connections

New in version 1.6.0.

You can specify a timeout in milliseconds for establishing a TCP connection. The default is infinity.

```
<option name="connect_timeout" value="5000" />
```

You can also change the timeout on a per-session basis using `set_option`.

```
<set_option name="connect_timeout" value="1000" />
```

You can also enable the TCP REUSEADDR option globally:

```
<option name="tcp_reuseaddr" value="true" />
```

IP transparent

New in version 1.7.0.

This option is used to set the `IP_TRANSPARENT` option on the TCP socket

```
<option name="ip_transparent" value="true" />
```

This can be useful to use when IPs are not configured on the client host (see also *iprange*)

Retry Attempts and Timeouts

New in version 1.6.0.

You can specify the amount of retry attempts made by Tsung. The default is 3.

```
<option name="max_retries" value="5" />
```

To disable retries entirely, set the value to 0.

In addition, the option `retry_timeout` (in milliseconds; defaults to 10) is used to implement a simple back-off algorithm (`retry * retry_timeout`).

```
<set_option name="retry_timeout" value="1000" />
```

Timeout for acknowledgments of messages

This is used to set the idle timeout(used for 'parse' and 'local' ack) and global ack timeout(used for 'global' ack). By default, idle timeout will be 10min(600000) and global ack timeout will be *infinity*. This value can be changed like this:

```
<option name="idle_timeout" value="300000"></option>
<option name="global_ack_timeout" value="6000000"></option>
```

Hibernate

New in version 1.3.1.

The option `hibernate` is used to reduced memory consumption of simulated users during thinktimes. By default, hibernation will be activated for thinktimes higher than 10sec. This value can be changed like this:

```
<option name="hibernate" value="5"></option>
```

To disable hibernation, you must set the value to *infinity*.

Rate_limit

New in version 1.4.0.

`rate_limit`. This will limit the bandwidth of each client (using a token bucket algorithm). The value is in KBytes per second. You can also specify a maximum burst value (eg. `max='2048'`). By default the burst size is the same as the rate (1024KB in the following example). Currently, only incoming traffic is rate limited.

```
<option name="rate_limit" value="1024"></option>
```

Ports_range

If you need to open more than 30000 simultaneous connections on a client machine, you will be limited by the number of TCP client ports, even if you use several IPs (this is true at least on Linux). To bypass this limit, Tsung must not delegate the selection of client ports and together with using several IP for each client, you have to defined a range for available clients ports, for ex:

```
<option name="ports_range" min="1025" max="65535"/>
```

Setting the seed for random numbers

If you want to use a fixed seed for the random generator, you can use the `seed` option, like this (by default, Tsung will use the current time to set the seed, therefore random numbers should be different for every test).

```
<option name="seed" value="42"/>
```

Path for BOSH

You can use the following config option for setting the path to BOSH request:

```
<option name="bosh_path" value="/http-bind"/>
```

Websocket options

When you use Websocket as a server type, you can set the following options for Websocket:

```
<option name="websocket_path" value="/chat"/>

<!-- send websocket data with text frame, default binary -->
<option name="websocket_frame" value="text"/>
<option name="websocket_subprotocols" value="chat, superchat"/>
```

Use `websocket_path` for setting the path of the websocket request; use `websocket_frame` for setting the frame type(option type: `binary` and `text`, and `binary` as default) of the sending websocket data. Use `websocket_subprotocols` for setting the `Sec-WebSocket-Protocol` header.

XMPP/Jabber options

Default values for specific protocols can be defined. Here is an example of option values for Jabber/XMPP:

```
<option type="ts_jabber" name="global_number" value="5" />
<option type="ts_jabber" name="userid_max" value="100" />
<option type="ts_jabber" name="domain" value="jabber.org" />
<option type="ts_jabber" name="username" value="myuser" />
<option type="ts_jabber" name="passwd" value="mypasswd" />
<option type="ts_jabber" name="muc_service" value="conference.localhost"/>
```

Using these values, users will be `myuserXXX` where `XXX` is an integer in the interval `[1:userid_max]` and `passwd` `mypasswdXXX`

If not set in the configuration file, the values will be set to:

- `global_number` = 100
- `userid_max` = 10000
- `domain` = `erlang-projects.org`
- `username` = `tsunguser`

- passwd = sesame

Other options are available if you prefer to use a CSV file for username/password, see [Reading usernames and password from a CSV file](#).

You can also set the muc_service here (see previous example).

HTTP options

For HTTP, you can set the UserAgent values (**available since Tsung 1.1.0**), using a probability for each value (the sum of all probabilities must be equal to 100)

```
<option type="ts_http" name="user_agent">
  <user_agent probability="80">
    Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.7.8) Gecko/20050513 Galeon/1.3.21
  </user_agent>
  <user_agent probability="20">
    Mozilla/5.0 (Windows; U; Windows NT 5.2; fr-FR; rv:1.7.8) Gecko/20050511 Firefox/
    ↪1.0.4
  </user_agent>
</option>
```

AMQP options

You can set the AMQP heartbeat timeout; for example to set it to 30s (default is 600s), add:

```
<option type="ts_amqp" name="heartbeat" value="30" />
```

Sessions

Sessions define the content of the scenario itself. They describe the requests to execute.

Each session has a given probability. This is used to decide which session a new user will execute. The sum of all session's probabilities must be 100.

Since Tsung 1.5.0, you can use weights instead of probabilities. In the following example, there will be twice as many sessions of type s1 than s2.

```
<session name="s1" weight="2" type="ts_http">
<session name="s2" weight="1" type="ts_http">
```

A transaction is just a way to have customized statistics. Say if you want to know the response time of the login page of your website, you just have to put all the requests of this page (HTML + embedded pictures) within a transaction. In the example above, the transaction called `index_request` will give you in the statistics/reports the mean response time to get `index.en.html` + `header.gif`. Be warn that If you have a thinktime inside the transaction, the thinktime will be part of the response time.

Thinktimes

You can set static or random thinktimes to separate requests. By default, a random thinktime will be a exponential distribution with mean equals to value.

```
<thinktime value="20" random="true"></thinktime>
```

In this case, the thinktime will be an exponential distribution with a mean equals to 20 seconds.

Since version 1.3.0, you can also use a range [min:max] instead of a mean for random thinktimes (the distribution will be uniform in the interval):

```
<thinktime min="2" max="10" random="true"></thinktime>
```

Since version 1.4.0, you can use a dynamic variable to set the thinktime value:

```
<thinktime value="%%_rndthink%" random="true"></thinktime>
```

You can also synchronize all users using the wait_global value:

```
<thinktime value='wait_global'>
```

which means: wait for all (N) users to be connected and waiting for the global lock (the value can be set using the option <option name="global_number" value = "XXX"/> and by setting *maxnumber=N* in <arrivalphase>).

Since version 1.6.0, you can wait for a ‘bidi’ ack. If your protocol is bidirectional (e.g. xmpp, websocket, ...), you can wait until the server sends some data, and the code that handle this data exits the think state.

```
<thinktime value="wait_bidi"></thinktime> -
```

HTTP

This example shows several features of the HTTP protocol support in Tsung: GET and POST request, basic authentication, transaction for statistics definition, conditional request (IF MODIFIED SINCE):

```
<sessions>
<session name="http-example" probability="70" type="ts_http">

  <request> <http url="/" method="GET" version="1.1">
    </http> </request>
  <request> <http url="/images/logo.gif"
    method="GET" version="1.1"
    if_modified_since="Fri, 14 Nov 2003 02:43:31 GMT">
    </http></request>

  <thinktime value="20" random="true"></thinktime>

  <transaction name="index_request">
    <request><http url="/index.en.html"
      method="GET" version="1.1" >
      </http> </request>
    <request><http url="/images/header.gif"
      method="GET" version="1.1">
      </http> </request>
  </transaction>

  <thinktime value="60" random="true"></thinktime>
  <request>
    <http url="/" method="POST" version="1.1"
      contents="bla=blu">
```

```
</http> </request>
<request>
  <http url="/bla" method="POST" version="1.1"
    contents="bla=blu&name=glop">
    <www_authenticate userid="Aladdin"
      passwd="open sesame"/></http>
  </request>
</session>

<session name="backoffice" probability="30" >
  <!-- -->
</session>
</sessions>
```

If you use an absolute URL, the server used in the URL will override the one specified in the `<server>` section. The following relative requests in the session will also use this new server value (until a new absolute URL is set).

New in 1.2.2: You can add any HTTP header now, as in:

```
<request>
  <http url="/bla" method="POST" contents="bla=blu&name=glop">
    <www_authenticate userid="Aladdin" passwd="open sesame"/>
    <http_header name="Cache-Control" value="no-cache"/>
    <http_header name="Referer" value="http://www.w3.org/" />
  </http>
</request>
```

New in 1.3.0: You can also read the content of a POST or PUT request from an external file:

```
<http url="mypage" method="POST" contents_from_file="/tmp/myfile" />
```

Since 1.3.1, you can also manually set a cookie, though the cookie is not persistent: you must add it in every `<requests>`:

```
<http url="/">
  <add_cookie key="foo" value="bar"/>
  <add_cookie key="id" value="123"/>
</http>
```

Authentication

Until Tsung 1.5.0, only Basic authentication was implemented. You can now use Digest Authentication and OAuth 1.0.

To use Digest authentication:

```
<!-- 1. First request return 401. We use dynvars to fetch nonce and realm -->
<request>
  <dyn_variable name="nonce" header="www-authenticate/nonce"/>
  <dyn_variable name="realm" header="www-authenticate/realm"/>
  <http url="/digest" method="GET" version="1.1"/>
</request>

<!--
2. This request will be authenticated. Type="digest" is important.
We use the nonce and realm values returned from the previous
If the webserver returns the nextnonce we set it to the nonce dynvar
```



```

for use with the next request.
Else it stays set to the old value
-->
<request subst="true">
  <dyn_variable name="nonce" header="authentication-info/nextnonce"/>
  <http url="/digest" method="GET" version="1.1">
    <www_authenticate userid="user" passwd="passwd" type="digest" realm="%%_realm%%"
↪nonce="%%_nonce%%"/>
  </http>
</request>

```

To use OAuth authentication:

```

<!-- Getting a Request Token -->

<request>
  <dyn_variable name="access_token" re="oauth_token=([^&]*)"/>
  <dyn_variable name="access_token_secret" re="oauth_token_secret=([^&]*)"/>
  <http url="/oauth/example/request_token.php" method="POST" version="1.1"
↪contents="empty">
    <oauth consumer_key="key" consumer_secret="secret" method="HMAC-SHA1"/>
  </http>
</request>

<!-- Getting an Access Token -->

<request subst='true'>
  <dyn_variable name="access_token" re="oauth_token=([^&]*)"/>
  <dyn_variable name="access_token_secret" re="oauth_token_secret=([^&]*)"/>
  <http url="/oauth/example/access_token.php" method="POST" version="1.1"
↪contents="empty">
    <oauth consumer_key="key" consumer_secret="secret" method="HMAC-SHA1"
↪access_token="%%_access_token%" access_token_secret="%%_access_token_secret%" />
  </http>
</request>

<!-- Making Authenticated Calls -->

<request subst="true">
  <http url="/oauth/example/echo_api.php" method="GET" version="1.1">
    <oauth consumer_key="key" consumer_secret="secret" access_token="%%_access_
↪token%" access_token_secret="%%_access_token_secret%" />
  </http>
</request>

```

Jabber/XMPP

Here is an example of a session definition for the Jabber/XMPP protocol:

```

<sessions>
  <session probability="70" name="jabber-example" type="ts_jabber">

    <request> <jabber type="connect" ack="local" /> </request>

    <thinktime value="2"></thinktime>
  </session>
</sessions>

```

```
<transaction name="authenticate">
  <request> <jabber type="auth_get" ack="local"></jabber> </request>
  <request> <jabber type="auth_set_plain" ack="local"></jabber> </request>
</transaction>

<request> <jabber type="presence:initial" ack="no_ack"/> </request>

<thinktime value="30"></thinktime>

<transaction name="online">
  <request> <jabber type="chat" ack="no_ack" size="16" destination="online"/></
↪request>
</transaction>

<thinktime value="30"></thinktime>

<transaction name="offline">
  <request> <jabber type="chat" ack="no_ack" size="56" destination="offline"/>
↪<request>
</transaction>

<thinktime value="30"></thinktime>

<transaction name="close">
  <request> <jabber type="close" ack="local"> </jabber></request>
</transaction>
</session>
</sessions>
```

Message stamping

It is possible to stamp chat message by setting stamped attribute of <jabber> element inside request to true. The stamp will include current timestamp and ID of the sender node. If the recipient will recognize the node ID, it will compare the timestamp inside message with the current one. The difference will be reported as xmpp_msg_latency metric (in milliseconds). The aim of node ID comparison is to avoid slight inconsistencies of timestamps on different Tsung nodes.

Only a fraction of requests will hit the same node they originated from, but with request rate high enough this fraction should be sufficient.

stamped is allowed only with size attribute. data will cause stamped to be ignored. There is a minimal length of the stamp, roughly 30 bytes. When size is greater than stamp length, random padding will be added to the stamp. If the stamp length is higher than size, then only stamp will be used as messagecontent, effectively exceeding specified length.

StartTLS

To secure a stream with STARTTLS, use:

```
<jabber type="starttls" ack="bidi_ack" />
```

Client certificate is implemented since **1.5.1**, for example, you can use dynamic variables like this:

```
<jabber type="starttls" ack="bidi_ack"
  cacertfile="%%_cacert%"
```

```
certfile="%_certfile%"
keyfile="%_keyfile%" />
```

Roster

What you can do with rosters using Tsung:

You can

1. Add a new contact to their roster - The new contact is added to the Tsung Group group, and their name matches their JID
2. Send a `subscribe` presence notification to the new contact's JID - This results in a *pending* subscription
3. Rename a roster contact This changes the previously added contact's name from the default JID, to Tsung Testuser
4. Delete the previously added contact.

Note that when you add a new contact, the contact JID is stored and used for the operations that follow. It is recommended that for each session which is configured to perform these operations, only do so once. In other words, you would NOT want to ADD more than one new contact per session. If you want to alter the rate that these roster functions are used during your test, it is best to use the session 'probability' factor to shape this.

The nice thing about this is that when you test run is complete, your roster tables should look the same as before you started the test. So, if you set it up properly, you can have pre-loaded roster entries before the test, and then use these methods to dynamically add, modify, and remove roster entries during the test as well.

Example roster modification setup:

```
<session probability="100" name="jabber-rostermod" type="ts_jabber">

  <!-- connect, authenticate, roster 'get', etc... -->

  <transaction name="rosteradd">
    <request>
      <jabber type="iq:roster:add" ack="no_ack" destination="online"></jabber>
    </request>
    <request>
      <jabber type="presence:subscribe" ack="no_ack"/>
    </request>
  </transaction>

  <!-- ... -->

  <transaction name="rosterrename">
    <request> <jabber type="iq:roster:rename" ack="no_ack"></jabber> </request>
  </transaction>

  <!-- ... -->

  <transaction name="rosterdelete">
    <request> <jabber type="iq:roster:remove" ack="no_ack"></jabber> </request>
  </transaction>

  <!-- remainder of session... -->

</session>
```

See also *Bidirectional Presence* for automatic handling of subscribing requests.

SASL Plain

SASL Plain authentication example:

```
<session probability="100" name="sasl" type="ts_jabber">

  <request> <jabber type="connect" ack="local"></jabber> </request>

  <thinktime value="10"></thinktime>

  <transaction name="authenticate">
    <request>
      <jabber type="auth_sasl" ack="local"></jabber></request>

      <request>
        <jabber type="connect" ack="local"></jabber> </request>

      <request>
        <jabber type="auth_sasl_bind" ack="local" ></jabber></request>
      <request>
        <jabber type="auth_sasl_session" ack="local" ></jabber></request>

    </transaction>
```

SASL Anonymous

SASL Anonymous authentication example:

```
<session probability="100" name="sasl" type="ts_jabber">

  <request> <jabber type="connect" ack="local"></jabber> </request>

  <thinktime value="10"></thinktime>

  <transaction name="authenticate">
    <request>
      <jabber type="auth_sasl_anonymous" ack="local"></jabber></request>

      <request>
        <jabber type="connect" ack="local"></jabber> </request>

      <request>
        <jabber type="auth_sasl_bind" ack="local" ></jabber></request>
      <request>
        <jabber type="auth_sasl_session" ack="local" ></jabber></request>

    </transaction>
```

Presence

- **type** can be either `presence:broadcast` or `presence:directed`.
- **show** value must be either `away`, `chat`, `dnd`, or `xa`.

- **status** value can be any text.

For more info, see section 2.2 of [RFC 3921](#).

If you omit the **show** or **status** attributes, they default to **chat** and **Available** respectively.

Example of broadcast presence (broadcast to members of your roster):

```
<request>
  <jabber type="presence:broadcast" show="away" status="Be right back..." ack="no_ack"
  ↪"/>
</request>

<thinktime value="5"></thinktime>

<request>
  <jabber type="presence:broadcast" show="chat" status="Available
  to chat" ack="no_ack"/>
</request>

<thinktime value="5"></thinktime>

<request>
  <jabber type="presence:broadcast" show="dnd" status="Don't bother me!" ack="no_ack"/
  ↪>
</request>

<thinktime value="5"></thinktime>

<request>
  <jabber type="presence:broadcast" show="xa" status="I may never come back..."
  ack="no_ack"/>
</request>

<thinktime value="5"></thinktime>

<request> <jabber type="presence:broadcast" ack="no_ack"/> </request>
<thinktime value="5"></thinktime>
```

Example of directed presence (sent to random online users):

```
<request>
  <jabber type="presence:directed" show="away" status="Be right back..." ack="no_ack"/
  ↪>
</request>

<thinktime value="5"></thinktime>

<request>
  <jabber type="presence:directed" show="chat" status="Available to chat" ack="no_ack
  ↪"/>
</request>

<thinktime value="5"></thinktime>

<request>
  <jabber type="presence:directed" show="dnd" status="Don't bother me!" ack="no_ack"/>
</request>

<thinktime value="5"></thinktime>

<request>
  <jabber type="presence:directed" show="xa" status="I may never come back..."
  ack="no_ack"/>
</request>
```

```
<thinktime value="5"></thinktime>

<request>
  <jabber type="presence:directed" ack="no_ack"/>
</request>
<thinktime value="5"></thinktime>
```

MUC

Tsung supports three MUC operations:

- Join a room (attribute type='muc:join')
- Send a message to a room (attribute type='muc:chat')
- Change nickname (attribute type='muc:nick')
- Exit a room (attribute type='muc:exit')

Here's an example:

```
<!-- First, choose an random room and random nickname: -->
<setdynvars sourcetype="random_number" start="1" end="100">
  <var name="room"/>
</setdynvars>
<setdynvars sourcetype="random_string" length="10">
  <var name="nick1"/>
</setdynvars>

<request subst="true">
  <jabber type='muc:join' ack="local" room="room%%_room%%" nick="%%_nick1%%"/>
</request>

<!-- use a for loop to send several messages to the room -->
<for from="1" to="6" var="i">
  <thinktime value="30"/>
  <request subst="true">
    <jabber type="muc:chat" ack="no_ack" size="16" room="room%%_room%%"/>
  </request>
</for>

<!-- change nickname-->
<thinktime value="2"/>
<setdynvars sourcetype="random_string" length="10">
  <var name="nick2"/>
</setdynvars>

<request subst="true">
  <jabber type="muc:nick" room="room%%_room%%" nick="%%_nick2%%"
    ack="no_ack"/>
</request>
```

MUC support is available since version **1.3.1**.

PubSub

Experimental support for PubSub is available in version **1.3.1**

You can read the following entry: <https://support.process-one.net/browse/TSUN-115>

VHost

VHost support is available since version **1.3.2**

Tsung is able to bench multiple vhost instances by choosing a vhost XMPP name from a list at connection time in the scenario.

The vhost list is read from a file:

```
<options>
...
<option name="file_server" value="domains.csv" id="vhostfileId"></option>
...
<option type="ts_jabber" name="vhost_file" value="vhostfileId"></option>
...
</options>
```

When each client starts a session, it chooses randomly a domain (each domain has the same probability).

Reading usernames and password from a CSV file

Since version 1.4.0, you can now use a CSV file to store the usernames and password.

Configure the CSV file:

```
<options>
  <option name="file_server" id='userdb' value="/home/foo/.tsung/users.csv"/>
</options>
```

And then you have to defined two variables of type file, and the first jabber request (connect) must include a `xmpp_authenticate` tag:

```
<session probability="100" name="jabber-example" type="ts_jabber">

  <setdynvars sourcetype="file" fileid="userdb" delimiter=";" order="iter">
    <var name="username" />
    <var name="password" />
  </setdynvars>

  <request subst='true'>
    <jabber type="connect" ack="no_ack">
      <xmpp_authenticate username="%%_username%%" passwd="%%_password%%"/>
    </jabber>
  </request>

  <thinktime value="2"></thinktime>

  <transaction name="authenticate">

    <request>
      <jabber type="auth_get" ack="local"> </jabber>
    </request>
    <request>
      <jabber type="auth_set_plain" ack="local"></jabber>
    </request>

  </transaction>

</session>
```

```
</transaction>
...
</session>
```

Moreover (since **1.5.0**), when using chat messages to random or offline users, you should disable the default users (not from CSV) by setting `userid_max` to 0 and by setting the `fileid` for offline and random users (also used for pubsub):

```
<options>
<option type="ts_jabber" name="userid_max" value="0" />
<option type="ts_jabber" name="random_from_fileid" value='userdb' />
<option type="ts_jabber" name="offline_from_fileid" value='userdb' />
<option type="ts_jabber" name="fileid_delimiter" value=";" />
</options>
```

The username (resp. passwd) should be the first (resp. second) entry in the each CSV line (the delimiter is by default `" ; "` and can be overridden).

raw XML

You can send raw XML data to the server using the `raw` type:

```
<jabber type="raw" ack="no_ack" data="&lt;stream&gt;foo&lt;/stream&gt;"></jabber>
```

Beware: you must encode XML characters like `<`, `>`, `&`, etc.

resource

By default, the XMPP resource is set to `tsung`. Since version 1.5.0, you can override this (in all `auth_*` and `register` requests) using the `resource` attribute.

PostgreSQL

For PostgreSQL, 4 types of requests are available:

- connect (to a given database with a given username)
- authenticate (with password or not)
- sql (basic protocol)
- close

In addition, the following parts of the extended protocol is supported:

- copy, copydone and copyfail
- parse, bind, execute, describe
- sync, flush

This example shows most of the features of a PostgreSQL session:

```
<session probability="100" name="pgsql-example" type="ts_pgsql">
  <transaction name="connection">
    <request>
      <pgsql type="connect" database="bench" username="bench" />
```



```

    </request>
</transaction>

<request><pgsql type="authenticate" password="sesame"/></request>

<thinktime value="12"/>

<request><pgsql type="sql">SELECT * from accounts;</pgsql></request>

<thinktime value="20"/>

<request><pgsql type="sql">SELECT * from users;</pgsql></request>

<request><pgsql type='sql'><![CDATA[SELECT n.nspname as "Schema",
c.relname as "Name",
CASE c.relkind WHEN 'r' THEN 'table' WHEN 'v' THEN 'view' WHEN 'i'
THEN 'index' WHEN 'S' THEN 'sequence' WHEN 's' THEN '%_toto_%' END as "Type",
u.username as "Owner"
FROM pg_catalog.pg_class c
LEFT JOIN pg_catalog.pg_user u ON u.usesysid = c.relowner
LEFT JOIN pg_catalog.pg_namespace n ON n.oid = c.relnamespace
WHERE c.relkind IN ('r','v','S','')
AND n.nspname NOT IN ('pg_catalog', 'pg_toast')
AND pg_catalog.pg_table_is_visible(c.oid)
ORDER BY 1,2;]]></pgsql></request>

<request><pgsql type="close"></pgsql></request>

</session>

```

Example with the extended protocol:

```

<request><pgsql type='parse' name_prepared='P0_7'><![CDATA[BEGIN;]]></pgsql></request>
<request><pgsql type='sync'></request>
<request><pgsql type='parse' name_prepared='P0_8'><![CDATA[UPDATE pgbench_accounts
SET abalance = abalance + $1 WHERE aid = $2;]]></pgsql></request>
<request><pgsql type='sync'></request>
<request><pgsql type='parse' name_prepared='P0_9'><![CDATA[SELECT
abalance FROM pgbench_accounts
WHERE aid = $1;]]></pgsql></request>
<request><pgsql type='sync'></request>
<request><pgsql type='parse' name_prepared='P0_10'><![CDATA[UPDATE pgbench_tellers
SET tbalance = tbalance + $1 WHERE tid = $2;]]></pgsql></request>
<request><pgsql type='sync'></request>
<request><pgsql type='parse' name_prepared='P0_11'><![CDATA[UPDATE pgbench_branches
SET bbalance = bbalance + $1 WHERE bid = $2;]]></pgsql></request>
<request><pgsql type='sync'></request>
<request><pgsql type='parse' name_prepared='P0_12'><![CDATA[INSERT
INTO pgbench_history (tid, bid, aid, delta, mtime)
VALUES ($1, $2, $3, $4, CURRENT_TIMESTAMP);]]></pgsql></request>
<request><pgsql type='sync'></request>
<request><pgsql type='parse' name_prepared='P0_13'><![CDATA[END;]]></pgsql></request>
<request><pgsql type='sync'></request>
<request><pgsql type='bind' name_prepared='P0_7' formats='none' formats_results='text
↵' /></request>
<request><pgsql type='describe' name_portal=''></request>
<request><pgsql type='execute'></request>
<request><pgsql type='sync'></request>

```

```
<request><pgsql type='bind' name_portal='' name_prepared='P0_8'
  formats='none' formats_results='text'
  parameters='2924,37801' /></request>
```

MySQL

For MySQL, 4 types of requests are available (same as PostgreSQL):

- connect (to a given database with a given username)
- authenticate (with password or not)
- sql
- close

This example shows most of the features of a MySQL session:

```
<session probability="100" name="mysql-example" type="ts_mysql">
<request>
  <mysql type="connect" />
</request>
<request>
  <mysql type="authenticate" database="test" username="test" password="test" />
</request>
<request>
  <mysql type="sql">SHOW TABLES</mysql>
</request>
<request>
  <mysql type="sql">SELECT * FROM mytable</mysql>
</request>
<request>
  <mysql type="close" />
</request>
</session>
```

Websocket

For Websocket, 3 types of requests are available:

- connect (to a given path)
- message (send message to server, add a attribute 'ack' to specify whether Tsung should wait for a response)
- close

Example with Websocket as a session type:

```
<session probability="100" name="websocket-example" type="ts_websocket">
  <request subst="true">
    <websocket type="connect" path="/path/to/ws"></websocket>
  </request>
  <request>
    <dyn_variable name="uid" jsonpath="uid"/>
    <!-- send data with text frame, default binary-->
    <websocket type="message" frame="text">{"user":"user", "password":"password"}</
↪websocket>
  </request>
```

```

<request subst="true">
  <match do="log" when="nomatch">ok</match>
  <websocket type="message">{"uid":"%_uid%", "data":"data"}</websocket>
</request>

<request>
  <websocket type="message" ack="no_ack">{"key":"value"}</websocket>
</request>

<request>
  <websocket type="close"></websocket>
</request>
</session>

```

You can do substitution on attribute ‘path’ and message content, match a response or define dynamic variables based on the response message.

You can also set the subprotocols in a connect message:

```

<websocket type="connect" path="/path/to/ws" subprotocols="chat"></websocket>

```

If you use `change_type` to start a websocket, don’t forget to set `bidi="true"`, like this:

```

<change_type new_type="ts_websocket" host="127.0.0.1" port="8080" server_type="tcp"
↪restore="true" store="true" bidi="true"/>i

```

When connecting to a websocket server you can set the `origin`, which can be checked by a websocket as a security measure, see <https://tools.ietf.org/html/rfc6455#section-10.2> for more details. If not set this defaults to the host value.

```

<websocket type="connect" origin="https://example.com"></websocket>

```

AMQP

For AMQP, it supports publish and consume messages on multiple channel, Available request types:

- `connection.open` (to a given vhost)
- `connection.close`
- `channel.open` (with specified and valid channel id)
- `channel.close` (with specified and valid channel id)
- `confirm.select` (on specified and already opened channel)
- `basic.qos` (on specified and already opened channel, only supports attribute ‘`prefetch_count`’)
- `basic.publish` (with channel id, exchange name, routing_key and the payload)
- `basic.consume` (with channel id, queue name)
- `waitForConfirms` (with timeout for confirmations from the server)
- `waitForMessages` (with timeout for messages delivered to the client)

Example with AMQP as a session type:

```

<session probability="100" name="amqp-example" type="ts_amqp" bidi="true">
  <request>
    <amqp type="connection.open" vhost="/"></amqp>
  </request>

  <!-- open channel, channel id is from 1 to 10 -->
  <for from="1" to="10" incr="1" var="loops">
    <request>
      <amqp type="channel.open"></amqp>
    </request>
  </for>

  <!-- ignore this request if you don't need publisher confirm -->
  <for from="1" to="10" incr="1" var="loops">
    <request subst="true">
      <amqp type="confirm.select" channel="%%_loops%%"></amqp>
    </request>
  </for>

  <for from="1" to="10" incr="1" var="loops">
    <for from="1" to="100" incr="1" var="counter">
      <transaction name="publish">
        <!-- specify payload_size to have tsung generate a payload for you -->
        <request subst="true">
          <amqp type="basic.publish" channel="%%_loops%%" exchange="test_
↪exchange"
          routing_key="test_queue" persistent="true" payload_size="100"></
↪amqp>
        </request>
        <!-- substitutions are supported on the payload. Payload will override_
↪payload_size. -->
        <request subst="true">
          <amqp type="basic.publish" channel="%%_loops%%" exchange="test_
↪exchange"
          routing_key="test_queue" persistent="true" payload="Test Payload">
↪</amqp>
        </request>
      </transaction>
    </for>

    <!-- if publish with confirm, add a waitForConfirms request as you need:
↪timeout=1s -->
    <request>
      <amqp type="waitForConfirms" timeout="1"></amqp>
    </request>
  </for>

  <for from="1" to="10" incr="1" var="loops">
    <request subst="true">
      <amqp type="basic.consume" channel="%%_loops%%" queue="test_queue" ack=
↪"true"></amqp>
    </request>
  </for>

  <!-- wait to receive messages from the server: timeout=180s -->
  <request>
    <amqp type="waitForMessages" timeout="180"></amqp>
  </request>

```

```

<for from="1" to="10" incr="1" var="loops">
  <request subst="true">
    <amqp type="channel.close" channel="%%_loops%%"></amqp>
  </request>
</for>

<request>
  <amqp type="connection.close"></amqp>
</request>
</session>

```

MQTT

It supports publish messages, subscribe and unsubscribe topics, Available request types:

- connect (with options like clean_start, will_topic, username, password, etc.)
- disconnect
- publish (with topic name, qos level and retain flag)
- subscribe (with topic name and qos level)
- unsubscribe (with topic name)
- waitForMessages (with timeout for messages published from server to the client)

Example with MQTT as a session type:

```

<session name="mqtt-example" probability="100" type="ts_mqtt">
  <request>
    <mqtt type="connect" clean_start="true" keepalive="10" will_topic="will_topic"
    ↪ will_qos="0" will_msg="will_msg" will_retain="false"></mqtt>
  </request>

  <for from="1" to="10" incr="1" var="loops">
    <request subst="true">
      <mqtt type="publish" topic="test_topic" qos="1" retained="true">test_
      ↪message</mqtt>
    </request>
  </for>

  <request subst="true">
    <mqtt type="subscribe" topic="test_topic" qos="1"></mqtt>
  </request>

  <request>
    <!-- wait for 60s -->
    <mqtt type="waitForMessages" timeout="60"></mqtt>
  </request>

  <request subst="true">
    <mqtt type="unsubscribe" topic="test_topic"></mqtt>
  </request>

  <request>
    <mqtt type="disconnect"></mqtt>
  </request>
</session>

```

```
</request>
</session>
```

LDAP

Authentication

The recommended mechanism used to authenticate users against a LDAP repository requires two steps to follow. Given an username and password, we:

- Search the user in the repository tree, using the username (so users can reside in different subtrees of the organization)
- Try to bind as the user, with the distinguished name found in the first step and the user's password

If the bind is successful, the user is authenticated (this is the scheme used, among others, by the LDAP authentication module for Apache http://httpd.apache.org/docs/2.0/mod/mod_auth_ldap.html)

LDAP Setup

For this example we are going to use a simple repository with the following hierarchy:

The repository has users in two organizational units

1. users (with four members)
2. users2 (with tree members)

For simplicity we set the password of each user to be the same as its common name (cn). Tsung Setup We will use a CSV file as input, containing the user:password pairs for our test. So we start by writing it, in this case we name the file `users.csv`:

```
user1;user1
user2;user2
user3;user3
user4;user4
jane;jane
mary;mary
paul;pablo
paul;paul
```

The pair `paul:pablo` should fail to authenticate, we will note that in the Tsung report. Then, in our Tsung scenario, we let Tsung know about this file:

```
<options>
  <option name="file_server" id="users" value="users.csv"/>
</options>
<!-- We use two dynamic variables to hold the username and password -->
<setdynvars sourcetype="file" fileid="users" delimiter=";" order="iter">
  <var name="username" />
  <var name="password" />
</setdynvars>
```

To start the authentication process we instruct Tsung to perform a search, to find the distinguished name of the user we are trying to authenticate

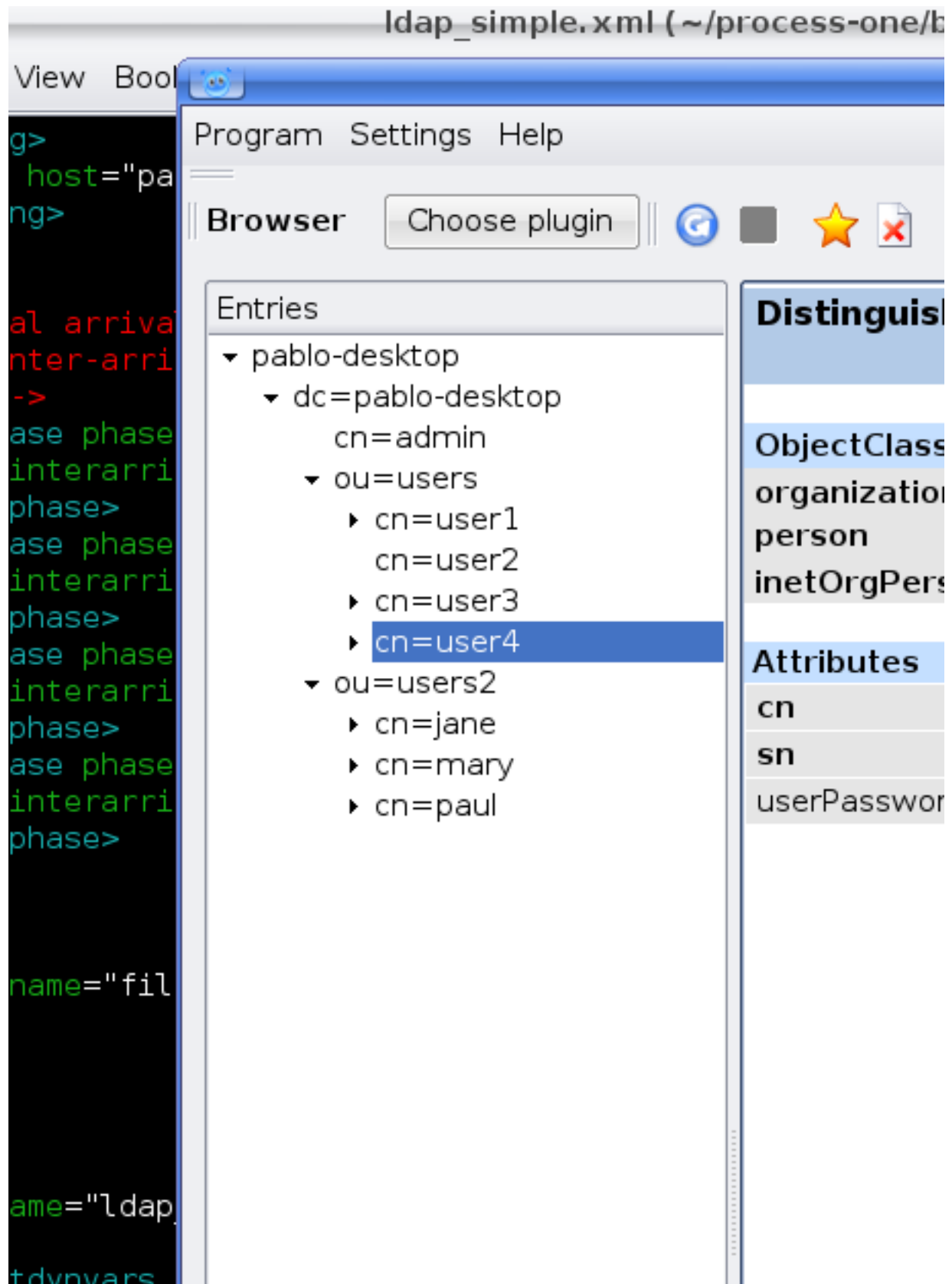


Fig. 6.1: LDAP hierarchy

```
<ldap type="search" base="dc=pablo-desktop" filter="(cn=%_username%)"
      result_var="search_result" scope="wholeSubtree"></ldap>
```

As we need to access the search result, we specify it using the `result_var` attribute. This attribute tells Tsung in which dynamic variable we want to store the result (if the `result_var` attribute isn't set, Tsung doesn't store the search result in any place). Finally, we try to bind as that user.

```
<request subst="true">
  <ldap type="bind" user="%%ldap_auth:user_dn%"
        password="%%_password%"></ldap>
</request>
```

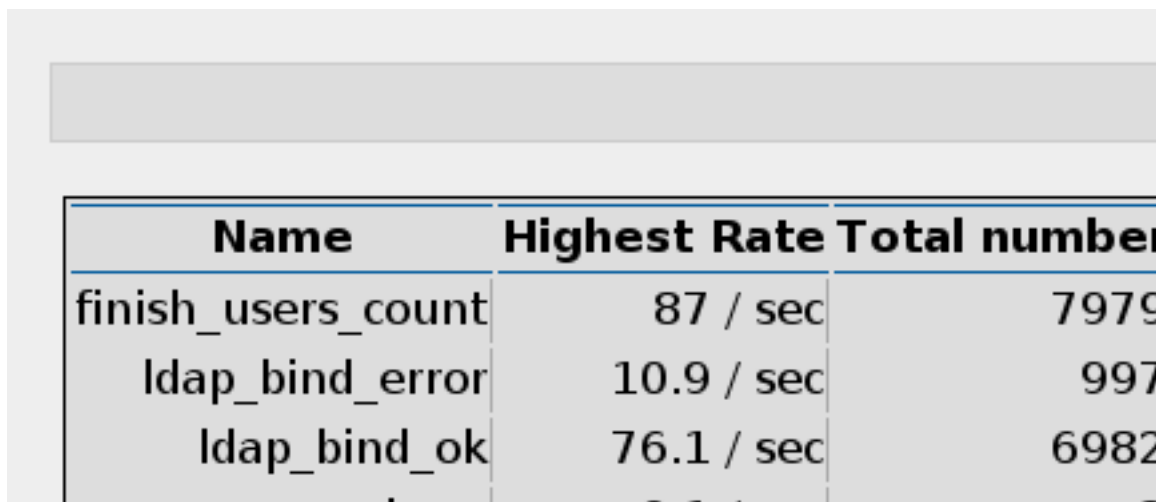
The only thing that remains to do is to implement the `ldap_auth:user_dn` function, that extract the distinguished name from the search result.

```
-module(ldap_auth).
-export([user_dn/1]).
user_dn({_Pid,DynVars}) ->
  [SearchResultEntry] = proplists:get_value(search_result,DynVars),
  {_,DN,_} = SearchResultEntry,
  DN.
```

We aren't covering errors here. supposing that there is always one (and only one) user found, that we extract from the `search_result` variable (as defined in the previous search operation). Each entry in the result set is a `SearchResultEntry` record. The record definition can be found in `<TSUNG_DIR>/include/ELDAPv3.hrl`.

As we only need to access the distinguished name of the object, we index into the result tuple directly. But if you need to access other attributes you probably will want to include the appropriate `.hrl` and use the record syntax instead. One of the eight `user:password` pairs in our `users` file was wrong, so we expect 1/8 of the authentication attempts to fail.

Indeed, after running the scenario we can confirm this in the Tsung report (see figure *LDAP Results*). The bind operation maintains two counters: `ldap_bind_ok` and `ldap_bind_error`, that counts successful and unsuccessful bind attempts.



Name	Highest Rate	Total number
finish_users_count	87 / sec	7979
ldap_bind_error	10.9 / sec	997
ldap_bind_ok	76.1 / sec	6982

Fig. 6.2: LDAP Results

Other examples

```
<session probability="100" name="ldap-example" type="ts_ldap">
  <request>
    <ldap type="bind" user="uid=foo" password="bar"/>
  </request>

  <request>
    <ldap type="search" base="dc=pablo-desktop" filter="(cn=user2)"
      scope="wholeSubtree"></ldap>
  </request>

  <!-- Add. Adds a new entry to the directory* -->
  <request subst="true">
    <ldap type="add" dn="%%_new_user_dn%%" >
      <attr type="objectClass">
        <value>organizationalPerson</value>
        <value>inetOrgPerson</value>
        <value>person</value>
      </attr>
      <attr type="cn"><value>%%_new_user_cn%%</value></attr>
      <attr type="sn"><value>fffs</value></attr>
    </ldap>
  </request>

  <!-- Modify. Modifies an existing entry; type=add/delete/modify-->
  <request subst="false">
    <ldap type="modify" dn="cn=u119843,dc=pablo-desktop" >
      <modification type="replace">
        <attr type="sn"><value>SomeSN</value></attr>
        <attr type="mail"><value>some@mail.com</value></attr>
      </modification>
    </ldap>
  </request>
</session>
```

Mixing session type

Since version **1.3.2**, a new tag **change_type** can be used in a session to change it's type.

```
<request>
  <jabber type="chat" ack="no_ack" size="16"
    destination="offline"/>
</request>

<thinktime value="3"/>

<change_type new_type="ts_http" host="foo.bar" port="80"
  server_type="tcp" store="true"/>

<request> <http url="http://foo.bar/"> </request>
<request> <http url="/favicon/"> </request>

<change_type new_type="ts_jabber" host="localhost" port="5222"
  server_type="tcp" restore="true"/>
```

```
<request> <jabber type="chat" ack="no_ack" size="16"
destination="previous"/> </request>
```

store="true" can be used to save the current state of the session (socket, cookies for http, ...) and restore="true" to reuse the previous state when you switch back to the old protocol.

You can use bidi="true" to indicate that the new protocol is bidirectional or bidi="false" for a non-bidirectional protocol (only available in version **1.5.1** and newer).

A dynamic variable set in the first part of the session will be available after a **<change_type>**. There is currently one caveat: you have to use a full URL in the first http request after a **<change_type>** (a relative URL will fail).

Raw

The **ts_raw** plugin allows you to send traffic to any kind of TCP/UDP server without any knowledge of the underlying protocol. You can set the data by attribute data, or just set a data size by attribute datasize (in this situation, Tsung send datasize bits of zeros). data and datasize can be a dynamic values.

The only way to control the response from the server is to use the ack attribute (also used by the **jabber** plugin):

- ack="local" as soon as a packet is received from the server, the request is considered as completed. Hence if you use a local ack with a request that does not require a response from the server, it will wait forever (or until a timeout is reached).
- ack="no_ack" as soon as the request is sent, it is considered as completed (do not wait for incoming data).
- ack="global" synchronized users. its main use is for waiting for all users to connect before sending messages. To do that, set a request with global ack (the value can be set using the option <option name="global_number" value="XXX"/> and by setting *maxnumber=N* in <arrivalphase>).

```
<session probability="100" name="raw" type="ts_raw">
  <transaction name="open">
    <request> <raw data="HELLO" ack="local"></raw> </request>
  </transaction>

  <thinktime value="4"/>
  <request> <raw datasize="2048" ack="local"></raw> </request>

  <transaction name="bye">
    <request> <raw data="BYEBYE" ack="local"></raw> </request>
  </transaction>
</session>
```

Advanced Features

Dynamic substitutions

Dynamic substitution are mark-up placed in element of the scenario. For HTTP, this mark-up can be placed in basic authentication (www_authenticate tag: userid and passwd attributes), URL (to change GET parameter) and POST content.

Those mark-up are of the form `%%Module:Function%%`. Substitutions are executed on a request-by-request basis, only if the request tag has the attribute subst="true".

When a substitution is requested, the substitution mark-up is replaced by the result of the call to the Erlang function: `Module:Function({Pid, DynData})` where `Pid` is the Erlang process id of the current virtual user and `DynData` the list of all Dynamic variables (**Warn: before version 1.1.0, the argument was just the Pid!**).

Here is an example of use of substitution in a Tsung scenario:

```
<session name="rec20040316-08:47" probability="100" type="ts_http">
  <request subst="true">
    <http url="/echo?symbol=%symbol:new%" method="GET"></http>
  </request>
</session>
```

For the http plugin, and since version 1.5.1, you can use the special value `subst='all_except_body'` instead of `'true'` to skip the substitutions in the body part of the HTTP response.

Here is the Erlang code of the module used for dynamic substitution:

```
-module(symbol).
-export([new/1]).

new({Pid, DynData}) ->
  case random:uniform(3) of
    1 -> "IBM";
    2 -> "MSFT";
    3 -> "RHAT"
  end.
```

Use `erlc` to compile the code, and put the resulting `.beam` file in `$PREFIX/lib/erlang/lib/tsung-X.X.X/ebin/` on all client machines.

As you can see, writing scenario with dynamic substitution is simple. It can be even simpler using dynamic variables (see later).

If you want to set unique id, you can use the built-in function `ts_user_server:get_unique_id`.

```
<session name="rec20040316-08:47" probability="100" type="ts_http">
  <request subst="true">
    <http url="/echo?id=%ts_user_server:get_unique_id%" method="GET" />
  </request>
</session>
```

Reading external file

New in 1.0.3: A new module `ts_file_server` is available. You can use it to read external files. For example, if you need to read user names and passwd from a CSV file, you can do it with it (currently, you can read only a single file).

You have to add this in the XML configuration file:

```
<option name="file_server" value="/tmp/userlist.csv"></option>
```

New in 1.2.2: You can read several files, using the `id` attribute to identify each file:

```
<option name="file_server" value="/tmp/userlist.csv"></option>
<option name="file_server" id='random' value="/tmp/randomnumbers.csv"></option>
```

Now you can build your own function to use it, for example, create a file called `readcsv.erl`:

```
-module(readcsv).  
-export([user/1]).  
  
user({Pid,DynVar})->  
    {ok,Line} = ts_file_server:get_next_line(),  
    [Username, Passwd] = string:tokens(Line,";"),  
    "username=" ++ Username ++ "&password=" ++ Passwd.
```

The output of the function will be a string `username=USER&password=PASSWORD`

Then compile it with `erlc readcsv.erl` and put `readcsv.beam` in `$prefix/lib/erlang/lib/tsung-VERSION/ebin` directory (if the file has an id set to random, change the call to `ts_file_server:get_next_line(random)`).

Then use something like this in your session:

```
<request subst="true">  
  </http>  
</request>
```

Two functions are available: `ts_file_server:get_next_line` and `ts_file_server:get_random_line`. For the `get_next_line` function, when the end of file is reached, the first line of the file will be the next line.

New in 1.3.0: you no longer have to create an external function to parse a simple csv file: you can use `setdynvars` (see next section for detailed documentation):

```
<setdynvars sourcetype="file" fileid="userlist.csv" delimiter=";" order="iter">  
  <var name="username" />  
  <var name="user_password" />  
</setdynvars>
```

This defines two dynamic variables **username** and **user_password** filled with the next entry from the csv file. Using the previous example, the request is now:

```
<request subst="true">  
  <http url="/login.cgi" version='1.0'  
    contents='username=%_username%&password=%_user_password%&op=login'  
    content_type='application/x-www-form-urlencoded' method='POST'>  
  </http>  
</request>
```

Much simpler than the old method!

In case you have several arrival phases programmed and if you use file with `order="iter"` the position in the file will not be reset between different arrival phase. You will not be returned to the first line when changing phase.

```
<arrivalphase phase="1" duration="10" unit="minute">  
  <users maxnumber="10" arrivalrate="100" unit="second" />  
</arrivalphase>  
<arrivalphase phase="2" duration="10" unit="minute">  
  <users maxnumber="20" arrivalrate="100" unit="second"></users>  
</arrivalphase>
```

In this example phase 1 will read about 10 lines and phase 2 will read the next 20 lines.

Dynamic variables

In some cases, you may want to use a value given by the server in a response later in the session, and this value is **dynamically generated** by the server for each user. For this, you can use `<dyn_variable>` in the scenario

Let's take an example with HTTP. You can easily grab a value in a HTML form like:

```
<form action="go.cgi" method="POST">
  <hidden name="random_num" value="42"></form>
</form>
```

with:

```
<request>
  <dyn_variable name="random_num"></dyn_variable>
  <http url="/testtsung.html" method="GET" version="1.0"></http>
</request>
```

Now `random_num` will be set to 42 during the users session. Its value will be replace in all mark-up of the form `%%_random_num%%` if and only if the request tag has the attribute `subst="true"`, like:

```
<request subst="true">
  <http url="/go.cgi" version="1.0"
    contents="username=nic&random_num=%%_random_num%%&op=login"
    content_type="application/x-www-form-urlencoded" method="POST">
  </http>
</request>
```

Regexp

If the dynamic value is not a form variable, you can set a regexp by hand, for example to get the title of a HTML page: the regexp engine uses the `re` module, a Perl like regular expressions module for Erlang.

```
<request>
  <dyn_variable name="mytitlevar"
    re="&lt;title&gt;(.*?)&lt;/title&gt;"/>
  <http url="/testtsung.html" method="GET" version="1.0"></http>
</request>
```

Previously (before 1.4.0), Tsung uses the old `regexp` module from Erlang. This is now deprecated. The syntax was:

```
<request>
  <dyn_variable name="mytitlevar"
    regexp="&lt;title&gt;(.*?)&lt;/title&gt;"/>
  <http url="/testtsung.html" method="GET" version="1.0"></http>
</request>
```

XPath

A new way to analyze the server response has been introduced in the release **1.3.0**. It is available only for the HTTP and XMPP plugin since it is based on XML/HTML parsing. This feature uses the `mochiweb` library and **only works with Erlang R12B and newer version**.

This give us some benefices:

- XPath is simple to write and to read, and match very well with HTML/XML pages

- The parser works on `binaries()`, and doesn't create any `string()`.
- The cost of parsing the HTML/XML and build the tree is amortized between all the `dyn_variables` defined for a given request

To utilize XPath expression, use a `xpath` attribute when defining the `dyn_variable`, instead of `re`, like:

```
<dyn_variable name="field1_value" xpath="//input[@name='field1']/@value"/>
<dyn_variable name="title" xpath="/html/head/title/text()" />
```

There is a bug in the XPath engine, result nodes from “descendant-or-self” aren't returned in document order. This isn't a problem for the most common cases.

However, queries like `//img[1]/@src` are not recommended, as the order of the `<img>` elements returned from `//img` is not the expected.

The order is respected for paths without “descendant-or-self” axis, so this: `/html/body/div[2]/img[3]/@src` is interpreted as expected and can be safely used.

It is possible to use XPath to get a list of elements from an html page, allowing dynamic retrieval of objects. You can either create embedded Erlang code to parse the list produced, or use `foreach` that was introduced in release **1.4.0**.

For XMPP, you can get all the contacts in a dynamic variable:

```
<request subst="true">
  <dyn_variable name="contactJids"
    xpath="//iq[@type='result']/query[@xmlns='jabber:iq:roster']//item[string-
    ↪length(@wr:type)=0]/@jid" />
  <jabber type="iq:roster:get" ack="local"/>
</request>
```

JSONPath

Another way to analyze the server response has been introduced in the release **1.3.2** when the server is sending JSON data. It is only for the HTTP plugin. This feature uses the `mochiweb` library and **only works with Erlang R13B and newer version**.

Tsung implements a (very) limited subset of JSONPath as defined here <http://goessner.net/articles/JsonPath/>

To utilize `jsonpath` expression, use a `jsonpath` attribute when defining the `<dyn_variable>`, instead of `re`, like:

```
<dyn_variable name="array3_value" jsonpath="field.array[3].value"/>
```

You can also use expressions `Key=Val`, e.g.:

```
<dyn_variable name="myvar" jsonpath="field.array[?name=bar].value"/>
```

PostgreSQL

New in version 1.3.2.

Since the PostgreSQL protocol is binary, `regexp` are not useful to parse the output of the server. Instead, a specific parsing can be done to extract content from the server's response; to do this, use the `pgsql_expr` attribute. Use `data_row[L][C]` to extract the column `C` of the line `L` of the data output. You can also use the literal name of the column (ie. the field name of the table). This example extract 3 dynamic variables from the server's response:

First one, extract the 3rd column of the fourth row, then the `mtime` field from the second row, and then it extract some data of the `row_description`.

```
<request>
  <dyn_variable name="myvar" pgsql_expr="data_row[4][3]" />
  <dyn_variable name="mtime" pgsql_expr="data_row[2].mtime" />
  <dyn_variable name="row" pgsql_expr="row_description[1][3][1]" />
  <pgsql type="sql">SELECT * from pgbench_history LIMIT 20;</pgsql>
</request>
```

A row description looks like this:

```
| =INFO REPORT==== 14-Apr-2010::11:03:22 ===
|         ts_pgsql:(7:<0.102.0>) PGSQL: Pair={row_description,
|                                     [{ "tid",text,1,23,4,-1,16395},
|                                     { "bid",text,2,23,4,-1,16395},
|                                     { "aid",text,3,23,4,-1,16395},
|                                     { "delta",text,4,23,4,-1,16395},
|                                     { "mtime",text,5,1114,8,-1,16395},
|                                     { "filler",text,6,1042,-1,26,16395}]]
```

So in the example, the **row** variable equals “aid”.

Decoding variables

It's possible to decode variable that contains html entities encoded, this is done with **decode** attribute set to **html_entities**.

```
<request>
  <dyn_variable name="mytitlevar"
    re="&lt;title&gt;(.*)&lt;/title&gt;"
    decode="html_entities" />
  <http url="/testtsung.html" method="GET" version="1.0"></http>
</request>
```

set_dynvars

Since version 1.3.0, more powerful dynamic variables are implemented.

You can set dynamic variables not only while parsing server data, but you can build them using external files or generate them with a function or generate random numbers/strings:

Several types of dynamic variables are implemented (sourcetype attribute):

- Dynamic variables defined by calling an Erlang function:

```
<setdynvars sourcetype="erlang" callback="ts_user_server:get_unique_id">
  <var name="idl" />
```

- Dynamic variables defined by parsing an external file:

```
<setdynvars sourcetype="file" fileid="userdb" delimiter=";" order="iter">
  <var name="user" />
  <var name="user_password" />
</setdynvars>
```

delimiter can be any string, and *order* can be **iter** or **random**

- A dynamic variable can be a random number (uniform distribution)

```
<setdynvars sourcetype="random_number" start="3" end="32">
  <var name="rndint" />
</setdynvars>
```

- A dynamic variable can be a random string

```
<setdynvars sourcetype="random_string" length="13">
  <var name="rndstring1" />
</setdynvars>
```

- A dynamic variable can be a urandom string: this is much faster than the random string, but the string is not really random: the same set of characters is always used.
- A dynamic variable can be generated by dynamic evaluation of erlang code:

```
<setdynvars sourcetype="eval"
  code="fun ({Pid,DynVars}) ->
    {ok,Val}=ts_dynvars:lookup(md5data,DynVars),
    ts_digest:md5hex(Val) end.">
  <var name="md5sum" />
</setdynvars>
```

In this case, we use tsung function `ts_dynvars:lookup` to retrieve the dynamic variable named `md5data`. This `dyn_variable` `md5data` can be set in any of the ways described in the Dynamic variables section *Dynamic variables*.

- A dynamic variable can be generated by applying a JSONPath specification (see *JSONPath*) to an existing dynamic variable:

```
<setdynvars sourcetype="jsonpath" from="notification" jsonpath="result[?state=OK].
↪node">
  <var name="deployed" />
</setdynvars>
```

- You can create dynamic variables to get the hostname and port of the current server

```
<setdynvars sourcetype="server">
  <var name="host" />
  <var name="port" />
</setdynvars>
```

- You can define a dynamic variable as constant value to use it in a plugin (since version **1.5.0**)

```
<setdynvars sourcetype="value" value="foobar">
  <var name="constant" />
</setdynvars>
```

A `setdynvars` can be defined anywhere in a session.

Checking the server's response

With the tag `match` in a `<request>` tag, you can check the server's response against a given string, and do some actions depending on the result. In any case, if it matches, this will increment the `match` counter, if it does not match, the `nomatch` counter will be incremented.

For example, let's say you want to test a login page. If the login is ok, the server will respond with `Welcome !` in the HTML body, otherwise not. To check that:


```
<request>
  <match do="continue" when="match">Welcome !</match>
  <http url="/login.php" version="1.0" method="POST"
    contents="username=nic&user_password=sesame"
    content_type="application/x-www-form-urlencoded" >
</request>
```

You can use a regexp instead of a simple string.

The list of available actions to do is:

- **continue**: do nothing, continue (only update match or nomatch counters)
- **log**: log the request id, userid, sessionid, name in a file (in `match.log`)
- **abort**: abort the session
- **abort_test**: abort the whole test
- **restart**: restart the session. The maximum number of restarts is 3 by default.
- **loop**: repeat the request, after 5 seconds. The maximum number of loops is 20 by default.
- **dump**: dump the content of the response in a file. The filename is `match-<userid>-<sessionid>-<requestid>-<dumpid>.dump`

You can mixed several match tag in a single request:

```
<request>
  <match do="loop" sleep_loop="5" max_loop="10" when="match">Retry</match>
  <match do="abort" when="match">Error</match>
  <http url='/index.php' method=GET'>
</request>
```

You can also do the action on **nomatch** instead of **match**.

If you want to skip the HTTP headers, and match only on the body, you can use **skip_headers='http'**. Also, you can apply a function to the content before matching; for example the following example use both features to compute the md5sum on the body of a HTTP response, and compares it to a given value:

```
<match do='log' when='nomatch' skip_headers='http' apply_to_content='ts_digest:md5hex
→ '>01441debe3d7cc65ba843eee1acff89d</match>
<http url="/" method="GET" version="1.1"/>
```

You can also use dynamic variables, using the **subst** attribute:

```
<match do='log' when='nomatch' subst='true' >%%_myvar%%</match>
<http url="/" method="GET"/>
```

Since 1.5.0, it's now possible to add **name** attribute in **match** tag to name a record printed in `match.log` as follow:

```
<match do='log' when='match' name='http_match_200ok'>200OK</match>
<http url="/" method="GET" version="1.1"/>
```

Loops, If, Foreach

Since 1.3.0, it's now possible to add conditional/unconditional loops in a session.

Since 1.4.0, it is possible to loop through a list of dynamic variables thanks to `foreach`.

<for>

Repeat the enclosing actions a fixed number of times. A dynamic variable is used as counter, so the current iteration could be used in requests. List of attributes:

from Initial value

to Last value

incr Amount to increment in each iteration

var Name of the variable to hold the counter

```
<for from="1" to="10" incr="1" var="counter">
  ...
  <request> <http url="/page?id=%_counter%"></http> </request>
  ...
</for>
```

<repeat>

Repeat the enclosing action (while or until) some condition. This is intended to be used together with <dyn_variable> declarations. List of attributes:

name Name of the repeat

max_repeat Max number of loops (default value is 20)

The last element of repeat must be either <while> or <until> example:

```
<repeat name="myloop" max_repeat="40">
  ...
  <request>
    <dyn_variable name="result" re="Result: (.*)" />
    <http url="/random" method="GET" version="1.1"></http>
  </request>
  ...
  <until var="result" eq="5" />
</repeat>
```

Since 1.3.1, it's also possible to add if statements based on dynamic variables:

<if>

```
<if var="tsung_userid" eq="3">
  <request> <http url="/foo"/> </request>
  <request> <http url="/bar"/> </request>
</if>
```

You can use eq or neq to check the variable.

Since 1.5.1 you can also use the comparison operators gt, gte, lt and lte to do respectively greater than, greater than or equal to, less than and less than or equal to.

If the dynamic variable is a list (output from XPath for example), you can access to the n-th element of a list like this:

```
<if var="myvar[1]" eq="3">
```

Here we compare the first element of the list to 3.

<abort>

Since 1.7.0 you can abort the session or the whole test by using an `<abort/>` element in a session (can be used inside an `<if>` statement for example). By default it will abort the current user session, but you can abort the whole test by setting the *type* attribute to *all* `<abort type='all' />`

<foreach>

Repeat the enclosing actions for all the elements contained in the list specified. The basic syntax is as follows:

```
<foreach name="element" in="list">
  <request subst="true">
    <http url="%%_element%" method="GET" version="1.1"/>
  </request>
</foreach>
```

It is possible to limit the list of elements you're looping through, thanks to the use of the *include* or *exclude* attributes inside the *foreach* statement.

As an example, if you want to include only elements with a local path you can write:

```
<foreach name="element" in="list" include="^/.*$">
```

If you want to exclude all the elements from a specific URI, you would write:

```
<foreach name="element" in="list" exclude="http://.*\.tld\.com\/.*$">
```

You can combine this with a XPath query. For instance the following scenario will retrieve all the images specified on a web page:

```
<request subst="true">
  <dyn_variable name="img_list" xpath="//img/@src"/>
  <http url="/mypage.html" method="GET" version="1.1"/>
</request>
<foreach name="img" in="img_list">
  <request subst="true">
    <http url="%%_img%" method="GET" version="1.1"/>
  </request>
</foreach>
```

Rate limiting

Since version 1.4.0, rate limiting can be enabled, either globally (see *Setting options*), or for each session separately.

For example, to limit the rate to 64KB/sec for a given session:

```
<session name="http-example" probability="70" type="ts_http">
  <set_option name="rate_limit" value="64" />
  ...
</session>
```

Only the incoming traffic is rate limited currently.

Requests exclusion

New in version 1.5.1.

It is possible to exclude some request for a special run. To do this you have to tag them and use the option `-x` when launching the run.

For example, to exclude the GET of `foo.png`, add a `tag` to the respective request:

```
<request>
  <http url="/" method="GET"></http>
</request>
<request tag="image">
  <http url="/foo.png" method="GET"></http>
</request>
```

Then launch the run with:

```
tsung -f SCENARIO.xml -x image start
```

Only the GET to `/` will be performed.

Note that request tags also get logged on **`dumptraffic="protocol"`** (see *File structure*)

Client certificate

New in version 1.5.1.

It is possible to use a client certificate for ssl authentication. You can use dynamic variables to set some parameters of the certificate (and the key password is optional).

```
<session name="https-with-cert" probability="70" type="ts_http">

  <set_option name="certificate">
    <certificate cacertfile="/etc/ssl/ca.pem"
      keyfile="%_keyfile%" keypass="%_keypass%" certfile="/home/nobody/
↪.tsung/client.pem"/>
  </set_option>
```

Statistics and Reports

File format

By default, Tsung use its own format (see FAQ *What is the format of the stats file tsung.log?*).

Since version 1.4.2, you can configure Tsung to use a JSON format; however in this case, the tools `tsung_stats.pl` and `tsung_plotter` will not work with the JSON files.

To enable JSON output, use:

```
<tsung backend="json" ...>
```

Example output file with JSON:

```
{
  "stats": [
    {"timestamp": 1317413841, "samples": []},
    {"timestamp": 1317413851, "samples": [
      {"name": "users", "value": 0, "max": 0},
      {"name": "users_count", "value": 0, "total": 0},
      {"name": "finish_users_count", "value": 0, "total": 0}]],
    {"timestamp": 1317413861, "samples": [
      {"name": "users", "value": 0, "max": 1},
      {"name": "load", "hostname": "requiem", "value": 1, "mean":
        0.0, "stddev": 0, "max": 0.0, "min": 0.0, "global_mean": 0
        , "global_count": 0},
      {"name": "freemem", "hostname": "requiem", "value": 1, "mean":
        2249.32421875, "stddev": 0, "max": 2249.32421875, "min":
        2249.32421875, "global_mean": 0, "global_count": 0},
      {"name": "cpu", "hostname": "requiem", "value": 1, "mean":
        4.790419161676647, "stddev": 0, "max": 4.790419161676647, "min":
        4.790419161676647, "global_mean": 0, "global_count": 0},
      {"name": "session", "value": 1, "mean": 387.864990234375, "stddev":
        0, "max": 387.864990234375, "min": 387.864990234375
        , "global_mean": 0, "global_count": 0},
```

```
{ "name": "users_count", "value": 1, "total": 1},
{ "name": "finish_users_count", "value": 1, "total": 1},
{ "name": "request", "value": 5, "mean": 75.331787109375, "stddev":
  46.689242405019954, "max": 168.708984375, "min": 51.744873046875
  , "global_mean": 0 , "global_count": 0},
{ "name": "page", "value": 1, "mean": 380.7548828125, "stddev":
  0.0, "max": 380.7548828125, "min": 380.7548828125 , "global_mean":
  0 , "global_count": 0},
{ "name": "connect", "value": 1, "mean": 116.70703125, "stddev":
  0.0, "max": 116.70703125, "min": 116.70703125 , "global_mean": 0
  , "global_count": 0},
{ "name": "size_rcv", "value": 703, "total": 703},
{ "name": "size_sent", "value": 1083, "total": 1083},
{ "name": "connected", "value": 0, "max": 0}, { "name": "http_304", "value": 5, "total
↪": 5}}]}
```

Available stats

- `request` Response time for each request.
- `page` Response time for each set of requests (a page is a group of request not separated by a thinktime).
- `connect` Duration of the connection establishment.
- `reconnect` Number of reconnection.
- `size_rcv` Size of responses in bytes.
- `size_sent` Size of requests in bytes.
- `session` Duration of a user's session.
- `users` Number of simultaneous users (it's session has started, but not yet finished).
- `connected` number of users with an opened TCP/UDP connection (example: for HTTP, during a think time, the TCP connection can be closed by the server, and it won't be reopened until the thinktime has expired). **new in 1.2.2.**
- `custom transactions`

The mean response time (for requests, page, etc.) is computed every 10 sec (and reset). That's why you have the highest mean and lowest mean values in the Stats report. **Since version 1.3.0**, the mean for the whole test is also computed.

HTTP specific stats:

- counter for each response status (200, 404, etc.)

Jabber specific stats:

- `request_noack` Counter of `no_ack` requests. Since response time is meaningless with `no_ack` requests, we keep a separate stats for this. **new in 1.2.2.**
- `async_unknown_data_rcv` Only if `bidi` is true for a session. Count the number of messages received from the server without doing anything. **new in 1.2.2.**

- `async_data_sent` Only if `bidi` is true for a session. Count the number of messages sent to the server in response of a message received from the server. **new in 1.2.2.**

OS monitoring stats:

- `{load, <host>}` System load average during the last minute
- `{cpu, <host>}` CPU percentage (Maximum is 100%, ex: on dual core system, 100% means: both cores are 100% used)
- `{freemem, <host>}` Free Memory

Design

A bit of explanation on the design and internals of the statistics engine:

Tsung was designed to handle thousands of requests/sec, for very long period of times (several hours) so it do not write all data to the disk (for performance reasons). Instead it computes on the fly an estimation of the mean and standard variation for each type of data, and writes these estimations every 10 seconds to the disk (and then starts a new estimation for the next 10 sec). These computations are done for two kinds of data:

- `sample`, for things like response time
- `sample_counter` when the input is a cumulative one (number of packet sent for ex.).

There are also two other types of useful data (no averaging is done for those):

- `counter`: a simple counter, for HTTP status code for ex.
- `sum` for ex. the cumulative HTTP response's size (it gives an estimated bandwidth usage).

Generating the report

Since **version 1.6.0**, you can use the embedded web server started by the controller on port 8091. So for example if your controller is running on `node0`, use the URL <http://node0:8091/> in your browser. It will display the current status of Tsung (see *Dashboard*) and generate on the fly the report and graphs. There's also an option when you start Tsung to keep the controller alive, even when the test is finished, in order to use the embedded web server (see `-k` option). By default the web server will stop when the test is finished.

You can still generate the reports by manually during or after the tests:

cd to the log directory of your test (say `~/tsung/log/20040325-16:33/`) and use the script `tsung_stats.pl`:

```
/usr/lib/tsung/bin/tsung_stats.pl
```

Note: You can generate the statistics even when the test is running!

use `-help` to view all available options:

```
Available options:
  [--help] (this help text)
  [--verbose] (print all messages)
  [--debug] (print receive without send messages)
```

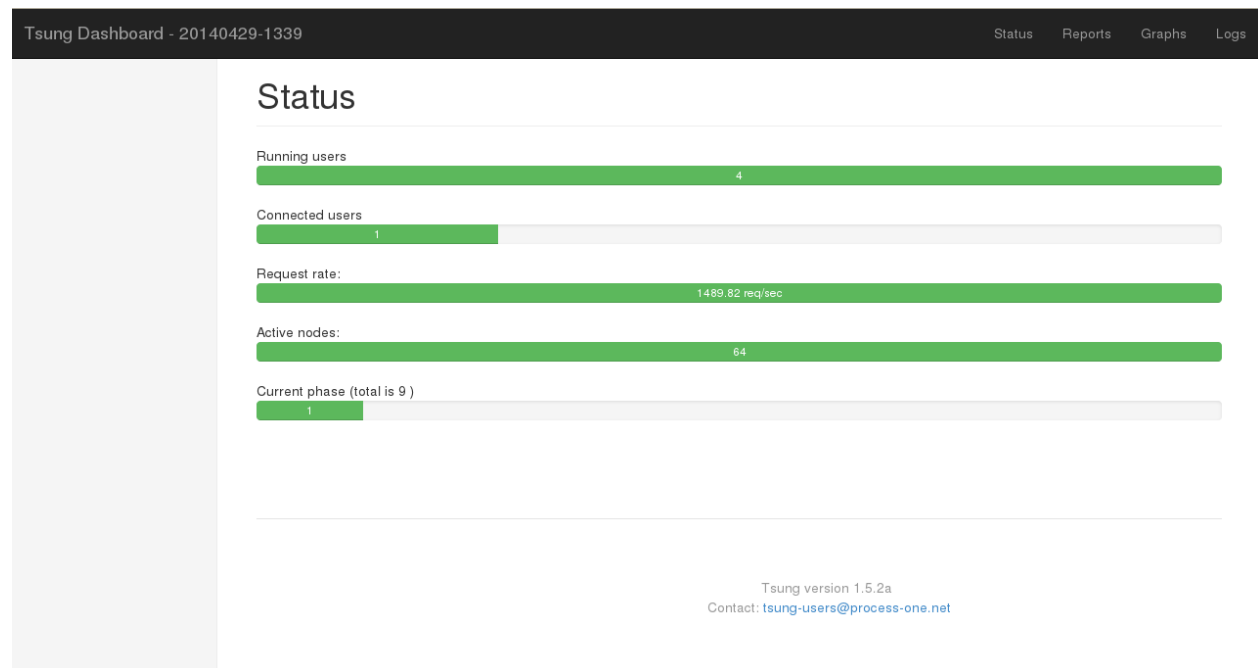


Fig. 7.1: Dashboard

```
[--dygraph] use dygraphs (http://dygraphs.com) to render graphs
[--noplot]  (don't make graphics)
[--gnuplot <command>] (path to the gnuplot binary)
[--nohtml]  (don't create HTML reports)
[--logy]   (logarithmic scale for Y axis)
[--tdir <template_dir>] (Path to the HTML tsung templates)
[--noextra] (don't generate graphics from extra data (os monitor, etc))
[--rotate-xtics] (rotate legend of x axes)
[--stats <file>] (stats file to analyse, default=tsung.log)
[--img_format <format>] (output format for images, default=png
                        available format: ps, svg, png, pdf)
```

Version **1.4.0** adds a new graphical output based on <http://dygraphs.com>.

Tsung summary

Figure *Report* shows an example of a summary report.

Graphical overview

Figure *Graphical output* shows an example of a graphical report.

Tsung

version 1.3.0

Stats Report

- [Main statistics](#)
- [Transactions](#)
- [Network](#)
- [Throughput](#)
- [Counters](#)
- [Server monitoring](#)
- [HTTP status](#)

Graphs Report

- [Response times](#)
- [Throughput graphs](#)
- [Simultaneous Users](#)
- [Server monitoring](#)
- [HTTP status](#)

XML Config file

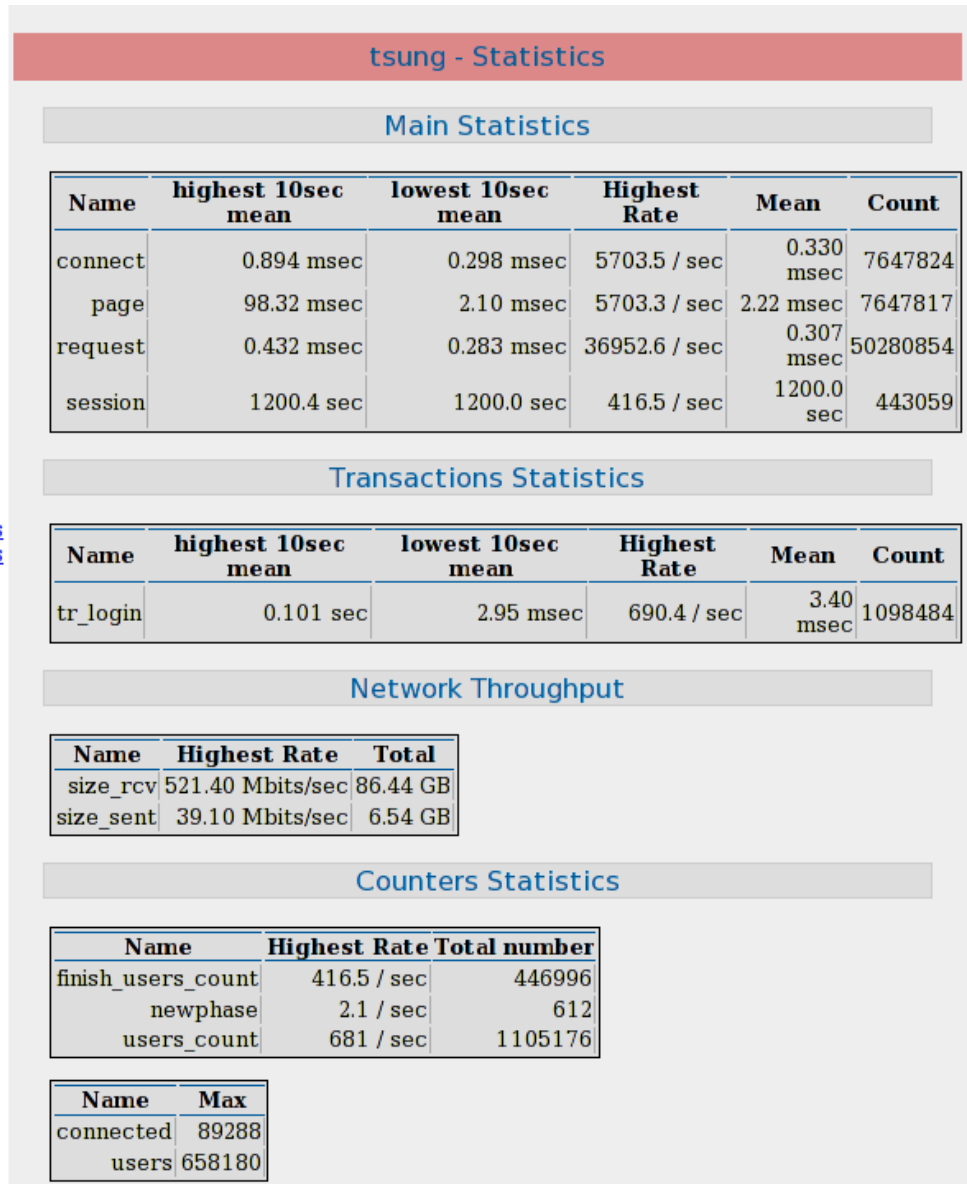


Fig. 7.2: Report

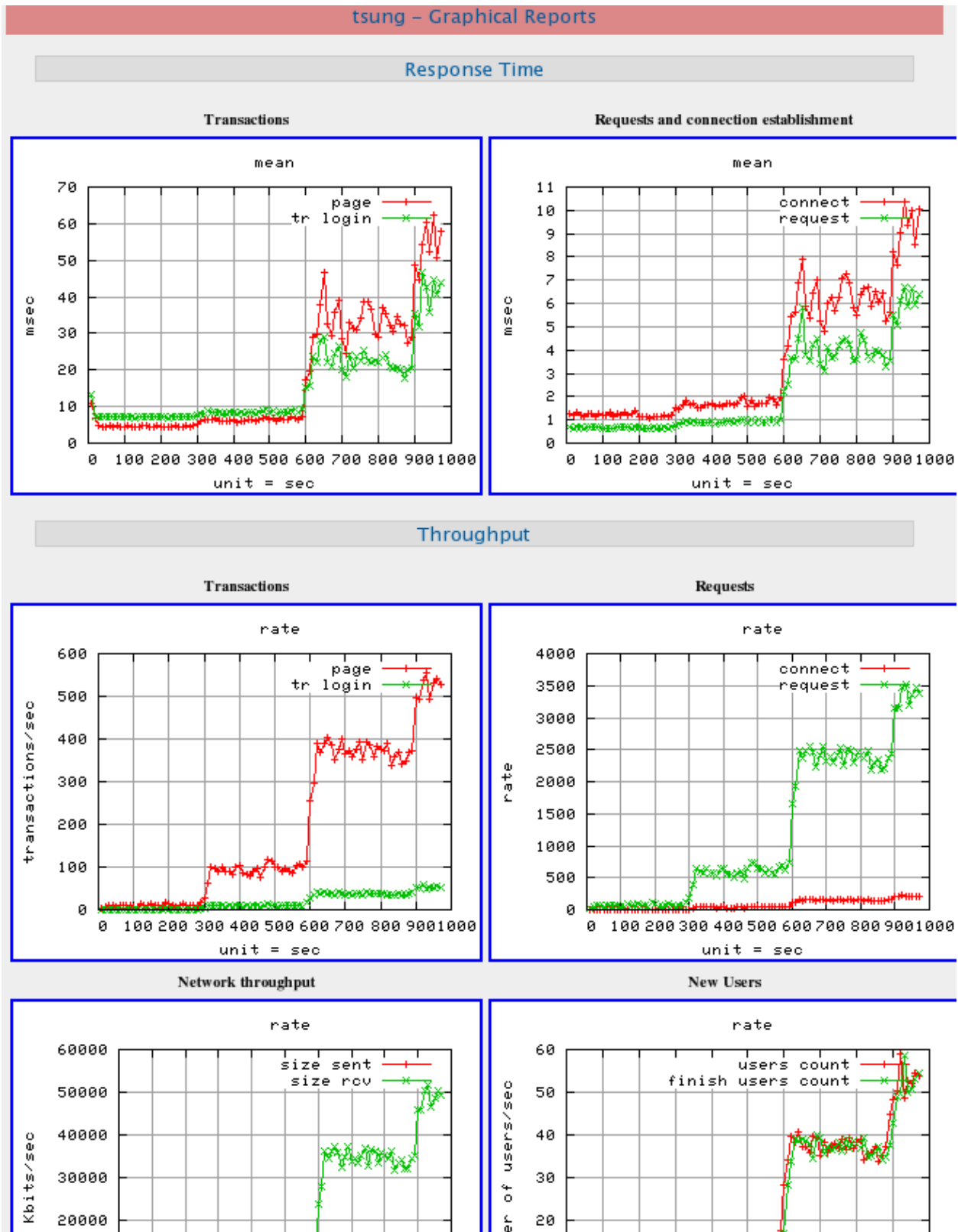


Fig. 7.3: Graphical output

Tsung Plotter

Tsung-Plotter (**tsplot** command) is an optional tool recently added in the Tsung distribution (it is written in Python), useful to compare different tests ran by Tsung. **tsplot** is able to plot data from several `tsung.log` files onto the same charts, for further comparisons and analyzes. You can easily customize the plots you want to generate by editing simple configuration files. You can get more information in the manual page of the tool (**man tsplot**).

Example of use:

```
tsplot "First test" firsttest/tsung.log "Second test" secondtest/tsung.log -d ↵
↵outputdir
```

Here's an example of the charts generated by `tsplot` (figure *Graphical output of tsplot*):

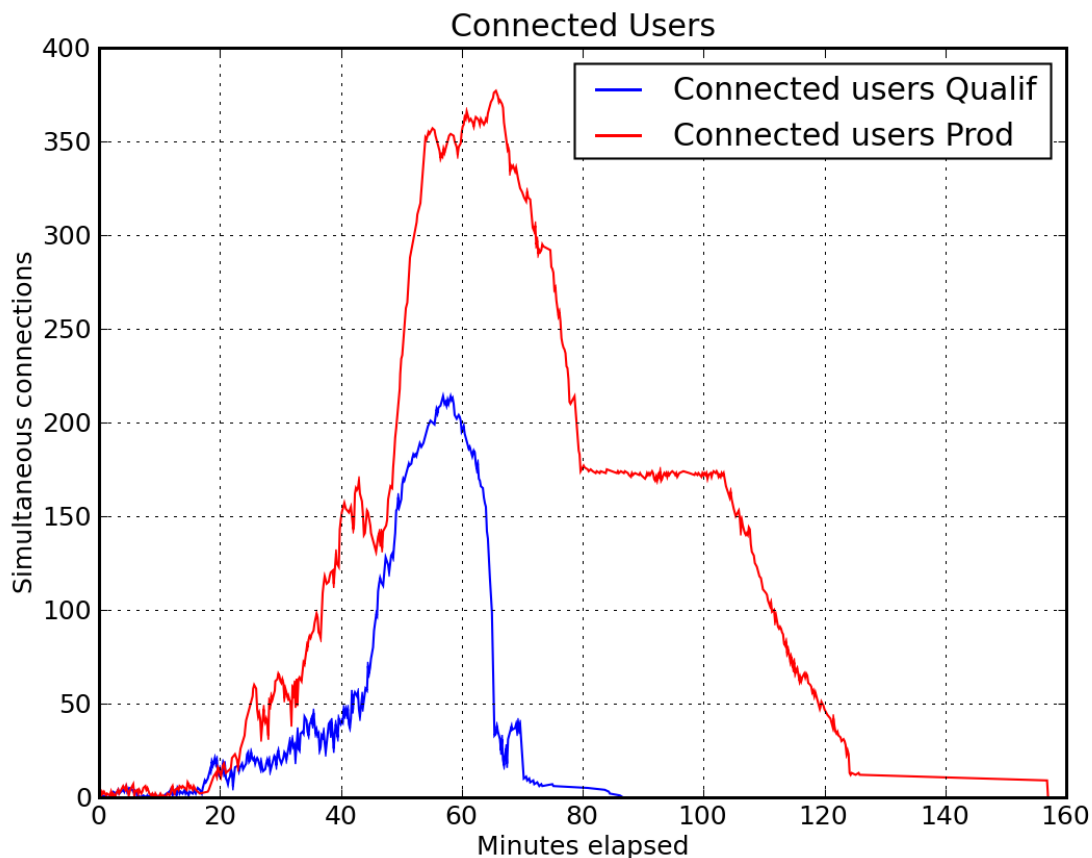


Fig. 7.4: Graphical output of `tsplot`

RRD

A contributed perl script **tsung-rrd.pl** is able to create rrd files from the Tsung log files. It's available in `/usr/lib/tsung/bin/tsung-rrd.pl`.

CHAPTER 8

References

- Tsung home page: <http://tsung.erlang-projects.org/>
- Tsung description (French)⁴
- Erlang web site <http://www.erlang.org/>
- Erlang programmation, Mickaël Rémond, Editions Eyrolles, 2003⁵
- **Making reliable system in presence of software errors**, Doctoral Thesis, Joe Armstrong, Stockholm, 2003⁶
- **Tutorial on How to write a Tsung plugin**, written by t ty, http://www.process-one.net/en/wiki/Writing_a_Tsung_plugin/

⁴ http://www.erlang-projects.org/Members/mremond/events/dossier_de_presentat/block_10766817551485/file

⁵ http://www.editions-eyrolles.com/php.accueil/Ouvrages/ouvrage.php3?ouv_ean13=9782212110791

⁶ http://www.sics.se/~joe/thesis/armstrong_thesis_2003.pdf

CHAPTER 9

Acknowledgments

The first version of this document was based on a talk given by Mickael Rémond² during an Object Web benchmarking workshop in April 2004 (more info at <http://jmob.objectweb.org/>).

² mickael.remond@erlang-fr.org

Frequently Asked Questions

Can't start distributed clients: timeout error

Most of the time, when a crash happened at startup without any traffic generated, the problem arise because the main Erlang controller node cannot create a “slave” Erlang virtual machine. The message looks like:

```
Can't start newbeam on host 'XXXXX (reason: timeout) ! Aborting!
```

The problem is that the Erlang slave module cannot start a remote slave node.

You can test this using this simple command on the controller node (remotehost is the name of the client node):

```
>erl -rsh ssh -sname foo -setcookie mycookie

Eshell V5.4.3 (abort with ^G)
(foo@myhostname)1>slave:start(remotehost,bar,"-setcookie mycookie").
```

You should see this:

```
{ok,bar@remotehost}
```

If you got {error,timeout}, it can be caused by several problems:

- ssh is not working (you must have a key without passphrase, or use an agent)
- Tsung and Erlang are not installed on all clients nodes
- Erlang version or location (install path) is not the same on all clients nodes
- A firewall is dropping Erlang packets: Erlang virtual machines use several TCP ports (dynamically generated) to communicate (if you are using EC2, you may have to change the Security Group that is applied on the VMs used for Tsung: open port range 0 - 65535)
- SELinux: You should disable SELinux on all clients.
- Bad /etc/hosts: This one is wrong (real hostname should not refer to localhost/loopback):

```
127.0.0.1 localhost myhostname
```

This one is good:

```
127.0.0.1 localhost
192.168.3.2 myhostname
```

- sshd configuration: For example, for SuSE 9.2 sshd is compiled with restricted set of paths (ie. when you shell into the account you get the users shell, when you execute a command via ssh you don't) and this makes it impossible to start an Erlang node (if Erlang is installed in `/usr/local` for example).

Run:

```
ssh myhostname erl
```

If the Erlang shell doesn't start then check what paths sshd was compiled with (in SuSE see `/etc/ssh/sshd_config`) and symlink from one of the approved paths to the Erlang executable (thanks to Gordon Guthrie for reporting this).

- old beam processes (Erlang virtual machines) running on client nodes: kill all beam processes before starting Tsung.

Note that you do not need to use the `127.0.0.1` address in the configuration file. It will not work if you use it as the injection interface. The shortname of your client machine should not refer to this address.

Warning Tsung launches a new Erlang virtual machine to do the actual injection even when you have only one machine in the injection cluster (unless `use_controller_vm` is set to true). This is because it needs to by-pass some limit with the number of open socket from a single process (1024 most of the time). The idea is to have several system processes (Erl beam) that can handle only a small part of the network connection from the given computer. When the `maxusers` limit (simultaneous) is reach, a new Erlang beam is launched and the newest connection can be handled by the new beam).

New in 1.1.0: If you don't use the distributed feature of Tsung and have trouble to start a remote beam on a local machine, you can set the `use_controller_vm` attribute to true:

```
<client host="mymachine" use_controller_vm="true">
```

Tsung crashes when I start it

Does your Erlang system has SSL support enabled ?

to test it:

```
> erl
Eshell V5.2 (abort with ^G)
1> ssl:start().
you should see 'ok'
```

Why do i have error_connect_emfile errors?

emfile error means : **too many open files**

This happens usually when you set a high value for `maxusers` (in the `<client>` section) (the default value is 800).

The errors means that you are running out of file descriptors; you must check that *maxusers* is less than the maximum number of file descriptors per process in your system (see `ulimit -n`).

You can either raise the limit of your operating system (see `/etc/security/limits.conf` for Linux) or decrease *maxusers* Tsung will have to start several virtual machine on the same host to bypass the *maxusers* limit.

It could be good if you want to test a large number of users to make some modifications to your system before launching Tsung:

- Put the domain name into `/etc/hosts` if you don't want the DNS overhead and you only want to test the target server
- Increase the maximum number of open files and customize TCP settings in `/etc/sysctl.conf`. For example:

```
net.ipv4.tcp_tw_reuse = 1
net.ipv4.tcp_tw_recycle = 1
net.ipv4.ip_local_port_range = 1024 65000
fs.file-max = 65000
```

Tsung still crashes/fails when I start it!

First look at the log file `~/.tsung/log/XXX/tsung_controller@yourhostname` to see if there is a problem.

If the file is not created and a crashed dump file is present, maybe you are using a binary installation of Tsung not compatible with the version of Erlang you used.

If you see nothing wrong, you can compile Tsung with full debugging: recompile with `make debug`, and don't forget to set the loglevel to debug in the XML file (see *tsung.xml log levels*).

To start the debugger or see what happen, start Tsung with the `debug` argument instead of `start`. You will have an Erlang shell on the `tsung_controller` node. Use `toolbar:start()` to launch the graphical tools provided by Erlang.

Can I dynamically follow redirect with HTTP?

If your HTTP server sends 30X responses (redirect) with dynamic URLs, you can handle this situation using a dynamic variable:

```
<request>
  <dyn_variable name="redirect" re="Location: (http://.*)\r"/>
  <http url="index.html" method="GET" ></http>
</request>

<request subst="true">
  <http url="%%_redirect%%" method="GET"></http>
</request>
```

You can even handle the case where the server use several redirections successively using a repeat loop (this works only with version 1.3.0 and up):

```
<request>
  <dyn_variable name="redirect" re="Location: (http://.*)\r"/>
  <http url="/test/redirect.html" method="GET"></http>
```

```
</request>

<repeat name="redirect_loop" max_repeat="5">
  <request subst="true">
    <dyn_variable name="redirect" re="Location: (http://.*)\r"/>
    <http url="%%_redirect%%" method="GET"></http>
  </request>
  <until var="redirect" eq=""/>
</repeat>
```

What is the format of the stats file tsung.log?

Sample tsung.log:

```
# stats: dump at 1218093520
stats: users 247 247
stats: connected 184 247
stats: users_count 184 247
stats: page 187 98.324 579.441 5465.940 2.177 9.237 595 58
stats: request 1869 0.371 0.422 5.20703125 0.115 0.431 7444062 581
stats: connect 186 0.427 0.184 4.47216796875 0.174 0.894 88665254 59
stats: tr_login 187 100.848 579.742 5470.223 2.231 56.970 91567888 58
stats: size_rcv 2715777 3568647
stats: 200 1869 2450
stats: size_sent 264167 347870
# stats: dump at 1218093530
stats: users 356 356
stats: users_count 109 356
stats: connected -32 215
stats: page 110 3.346 0.408 5465.940 2.177 77.234 724492 245
stats: request 1100 0.305 0.284 5.207 0.115 0.385 26785716 2450
stats: connect 110 0.320 0.065 4.472 0.174 0.540 39158164 245
stats: tr_login 110 3.419 0.414 5470.223 2.231 90.461 548628831 245
stats: size_rcv 1602039 5170686
stats: 200 1100 3550
stats: size_sent 150660 498530
...
```

the format is, for request, page, session and transactions tr_XXX:

```
stats: name, 10sec_count, 10sec_mean, 10sec_stddev, max, min, mean, count
```

or for HTTP returns codes, size_sent and size_rcv:

```
stats: name, count(during the last 10sec), totalcount(since the beginning)
```

How can I compute percentile/quartiles/median for transactions or requests response time?

It's not directly possible. But since **version 1.3.0**, you can use a new experimental statistic backend: set `backend="fullstats"` in the `<tsung>` section of your configuration file (also see [File structure](#)).

This will print every statistics data in a raw format in a file named `tsung-fullstats.log`. **Warning:** this may impact the performance of the controller node (a lot of data has to be written to disk).

The data looks like:

```
{sum,connected,1}
{sum,connected,-1}
[{sample,request,214.635},
 {sum,size_rcv,268},
 {sample,page,831.189},
 {count,200},
 {sum,size_sent,182},
 {sample,connect,184.787},
 {sample,request,220.974},
 {sum,size_rcv,785},
 {count,200},
 {sum,size_sent,164},
 {sample,connect,185.482}]
{sum,connected,1}
[{count,200},{sum,size_sent,161},{sample,connect,180.812}]
[{sum,size_rcv,524288},{sum,size_rcv,524288}]
```

Since version 1.5.0, a script `tsung_percentile.pl` is provided to compute the percentiles from this file.

How can I specify the number of concurrent users?

You can't. But it's on purpose: the load generated by Tsung is dependent on the arrival time between new clients. Indeed, once a client has finished his session in Tsung, it stops. So the number of concurrent users is a function of the arrival rate and the mean session duration.

For example, if your web site has 1,000 visits/hour, the arrival rate is $1000/3600 = 0.2778$ visits/second. If you want to simulate the same load, set the inter-arrival time is to $1/0.27778 = 3.6$ sec (e.g. `<users interarrival="3.6" unit="second">` in the `arrivalphase` node in the XML config file).

SNMP monitoring doesn't work?!

It use SNMP v1 and the "public" community. It has been tested with <http://net-snmp.sourceforge.net/>.

You can try with `snmpwalk` to see if your `snmpd` config is ok:

```
>snmpwalk -v 1 -c public IP-OF-YOUR-SERVER .1.3.6.1.4.1.2021.4.5.0
UCD-SNMP-MIB::memTotalReal.0 = INTEGER: 1033436
```

SNMP doesn't work with Erlang R10B and Tsung older than 1.2.0.

There is a small bug in the `snmp_mgr` module in old Erlang release (R9C-0). This is fixed in Erlang R9C-1 and up, but you can apply this patch to make it work on earlier version:

```
--- lib/snmp-3.4/src/snmp_mgr.erl.orig 2004-03-22 15:21:59.000000000 +0100
+++ lib/snmp-3.4/src/snmp_mgr.erl      2004-03-22 15:23:46.000000000 +0100
@@ -296,6 +296,10 @@
     end;
     is_options_ok([{recbuf,Sz}|Opts]) when 0 < Sz, Sz <= 65535 ->
         is_options_ok(Opts);
+is_options_ok([{receive_type, msg}|Opts]) ->
```

```
+    is_options_ok(Opts);
+is_options_ok([receive_type, pdu|Opts]) ->
+    is_options_ok(Opts);
+    is_options_ok([InvOpt|_]) ->
+        {error,{invalid_option,InvOpt}};
+is_options_ok([]) -> true.
```

How can i simulate a fix number of users?

Use `maxnumber` to set the max number of concurrent users in a phase, and if you want Tsung to behave like `ab`, you can use a loop in a session (to send requests as fast as possible); you can also define a max duration in `<load>`.

```
<load duration="5" unit="minute">
  <arrivalphase phase="1" duration="10" unit="minute">
    <users maxnumber="10" arrivalrate="100" unit="second"></users>
  </arrivalphase>
</load>
<sessions>
  <session probability="100" name="ab">
    <for from="1" to="1000" var="i">
      <request>
        <http url="http://myserver/index.html" method="GET"></http>
      </request>
    </for>
  </session>
</sessions>
```

error_closed

Only for non persistent session (XMPP); the server unexpectedly closed the connection; the session is aborted.

error_inet_<ERRORNAME>

Network error; see <http://www.erlang.org/doc/man/inet.html> for the list of all errors.

error_unknown_data

Data received from the server during a thinktime (not for unparsed protocol like XMPP). The session is aborted.

error_unknown_msg

Unknown message received (see the log files for more information). The session is aborted.

error_unknown

Abnormal termination of a session, see log file for more information.

error_repeat_<REPEATNAME>

Error in a repeat loop (undefined dynamic variable usually).

error_send_<ERRORNAME>

Error while sending data to the server, see <http://www.erlang.org/doc/man/inet.html> for the list of all errors.

error_send

Unexpected error while sending data to the server, see the logfiles for more information.

error_connect_<ERRORNAME>

Error while establishing a connection to the server. See <http://www.erlang.org/doc/man/inet.html> for the list of all errors.

error_no_online

XMPP: No online user available (usually for a chat message destined to a online user)

error_no_offline

XMPP: No offline user available (usually for a chat message destined to a offline user)

error_no_free_userid

For XMPP: all users Id are already used (`userid_max` is too low ?)

error_next_session

A clients fails to gets its session parameter from the `config_server`; the controller may be overloaded ?

error_mysql_<ERRNO>

Error reported by the mysql server (see <http://dev.mysql.com/doc/refman/5.0/en/error-messages-server.html>)

error_mysql_badpacket

Bad packet received for mysql server while parsing data.

error_pgsql

Error reported by the postgresql server.

CHAPTER 12

Changelog

CHAPTER 13

tsung-1.0.dtd

```
<?xml version="1.0" encoding="utf-8" ?>
<!ELEMENT tsung (information?, clients, servers, monitoring?, load, options?,
↪sessions)>

<!ELEMENT information (name|description|username|organisation)*>

<!ELEMENT name (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT username (#PCDATA)>
<!ELEMENT organisation (#PCDATA)>

<!ATTLIST tsung
    dumptraffic (true | false | light | protocol | protocol_local) "false"
    backend      (text | json| rrdtool | fullstats) "text"
    loglevel     (emergency|critical|error|warning|notice|info|debug) "notice"
    version      NMTOKEN #IMPLIED>

<!ELEMENT servers (server+)>
<!ELEMENT server EMPTY>
<!ATTLIST server
    host      NMTOKEN #REQUIRED
    port      NMTOKEN #REQUIRED
    weight    NMTOKEN "1"
    type      (ssl | tcp | udp | erlang | ssl6 | tcp6 | udp6 |bosh | bosh_ssl | websocket,
↪| websocket_ssl ) #REQUIRED>

<!ELEMENT clients (client+)>
<!ELEMENT client (ip*| iprange) >
<!ATTLIST client
    cpu      NMTOKEN "1"
    type      (machine | batch) "machine"
    host      NMTOKEN #IMPLIED
    batch     (torque | pbs | lsf | oar) #IMPLIED
    scan_intf NMTOKEN #IMPLIED
```

```
maxusers NMTOKEN "800"
use_controller_vm (true | false) "false"
weight NMTOKEN "1">

<!ELEMENT ip EMPTY>
<!ATTLIST ip
  value NMTOKEN #REQUIRED
  scan (true| false) "false"
>

<!ELEMENT iprange EMPTY>
<!ATTLIST iprange
  value NMTOKEN #REQUIRED
  version NMTOKEN "v4"
>

<!ELEMENT monitoring ( monitor+ )>
<!ELEMENT monitor ( snmp? | munin? | mysqladmin? )>
<!ATTLIST monitor
  host NMTOKEN #REQUIRED
  batch (true | false) "false"
  type (snmp | erlang | munin) "erlang">

<!ELEMENT mysqladmin EMPTY>
<!ATTLIST mysqladmin
  port NMTOKEN "3306"
  username NMTOKEN "root"
  password NMTOKEN #IMPLIED>

<!ELEMENT snmp (oid)*>
<!ATTLIST snmp
  version (v1 | v2) "v1"
  community NMTOKEN "public"
  port NMTOKEN "161">

<!ELEMENT oid EMPTY>
<!ATTLIST oid
  value NMTOKEN #REQUIRED
  name NMTOKEN #REQUIRED
  type NMTOKEN "sample"
  eval CDATA #IMPLIED>

<!ELEMENT munin EMPTY>
<!ATTLIST munin
  port NMTOKEN "4949">

<!ELEMENT load (arrivalphase | user)+>
<!ATTLIST load
  duration NMTOKEN #IMPLIED
  unit (hour | minute | second) "second"
  loop NMTOKEN "0"
>

<!ELEMENT arrivalphase (users | session_setup)+>
<!ATTLIST arrivalphase
  duration NMTOKEN #REQUIRED
  phase NMTOKEN #REQUIRED
  wait_all_sessions_end NMTOKEN "false"
  unit (hour | minute | second | millisecond) #REQUIRED>
```

```

<!ELEMENT users EMPTY>
<!ATTLIST users
    interarrival NMTOKEN #IMPLIED
    arrivalrate  NMTOKEN #IMPLIED
    unit (hour | minute | second) #REQUIRED
    maxnumber NMTOKEN #IMPLIED>

<!ELEMENT user EMPTY>
<!ATTLIST user
    start_time NMTOKEN #IMPLIED
    unit (hour | minute | second | millisecond) "second"
    session    CDATA #REQUIRED>

<!ELEMENT options (option*)>
<!ELEMENT option (user_agent*)>
<!ATTLIST option
    name      NMTOKEN #REQUIRED
    override (true | false) #IMPLIED
    random    (true | false) #IMPLIED
    id        NMTOKEN #IMPLIED
    min       NMTOKEN #IMPLIED
    max       NMTOKEN #IMPLIED
    type      (ts_http | ts_jabber | ts_pgsql | ts_amqp) #IMPLIED
    value     CDATA #IMPLIED>

<!ELEMENT set_option (user_agent*| certificate)>
<!ATTLIST set_option
    name      NMTOKEN #REQUIRED
    id        NMTOKEN #IMPLIED
    min       NMTOKEN #IMPLIED
    max       NMTOKEN #IMPLIED
    type      (ts_http | ts_jabber | ts_pgsql) #IMPLIED
    value     CDATA #IMPLIED>

<!ELEMENT certificate EMPTY >
<!ATTLIST certificate
    cacertfile CDATA #IMPLIED
    keyfile    CDATA #IMPLIED
    keypass    CDATA #IMPLIED
    certfile   CDATA #IMPLIED
>

<!ELEMENT sessions (session+)>
<!ELEMENT session ( request | thinktime | transaction | setdynvars | for |
repeat | if | change_type | foreach | set_option | interaction | abort )*>
<!ATTLIST session
    name      CDATA #REQUIRED
    bidi      CDATA #IMPLIED
    persistent (true | false) #IMPLIED
    probability NMTOKEN #IMPLIED
    weight     NMTOKEN #IMPLIED
    type       (ts_jabber | ts_http | ts_raw | ts_pgsql | ts_ldap | ts_webdav |ts_
mysql| ts_fs | ts_shell | ts_job | ts_websocket | ts_amqp | ts_mqtt) #REQUIRED>

<!ELEMENT session_setup EMPTY>
<!ATTLIST session_setup
    name      CDATA #REQUIRED

```

```

    probability NMTOKEN #IMPLIED
    weight      NMTOKEN #IMPLIED
>

<!--ELEMENT abort EMPTY-->
<!--ATTLIST abort
      type (session|all) "session" -->

<!--ELEMENT interaction EMPTY-->
<!--ATTLIST interaction
      action (send|receive) #REQUIRED
      id      CDATA #REQUIRED-->

<!--ELEMENT change_type EMPTY-->
<!--ATTLIST change_type
      new_type      (ts_jabber | ts_http | ts_raw | ts_pgsql | ts_ldap | ts_webdav_
→| ts_mysql | ts_fs | ts_shell | ts_job | ts_websocket | ts_amqp | ts_mqtt) #REQUIRED
      host CDATA #REQUIRED
      port CDATA #REQUIRED
      server_type NMTOKEN #REQUIRED
      store ( true | false ) "false"
      restore ( true | false ) "false"
      bidi ( true | false ) "false"
>

<!--ELEMENT request ( match*, dyn_variable*, ( http | jabber | raw |
      pgsql | ldap | mysql | fs | shell | job | websocket | amqp | mqtt) )-->
<!--ATTLIST request
      subst (true|false|all_except_body) "false"
      tag NMTOKEN "undefined"
>

<!--ELEMENT match (#PCDATA)-->
<!--ATTLIST match
      do (continue|loop|abort|restart|log|dump|abort_test) "continue"
      when (match|nomatch) "match"
      subst (true|false) "false"
      loop_back NMTOKEN "0"
      name NMTOKEN "-"
      max_loop NMTOKEN "20"
      max_restart NMTOKEN "3"
      sleep_loop NMTOKEN "5"
      apply_to_content NMTOKEN "undefined"
      skip_headers NMTOKEN "no"
>

<!--ELEMENT thinktime EMPTY-->
<!--ATTLIST thinktime
      random (true|false) "false"
      value CDATA #IMPLIED
      min NMTOKEN #IMPLIED
      max NMTOKEN #IMPLIED
>

<!--ELEMENT user_agent (#PCDATA)*-->
<!--ATTLIST user_agent
      probability NMTOKEN #REQUIRED
>

```



```

<!ELEMENT transaction (request | setdynvars | thinktime | for | repeat
    | if | foreach | interaction | abort )+>
<!ATTLIST transaction name NMTOKEN #REQUIRED>

<!ELEMENT http (oauth?, www_authenticate?, soap?, http_header*, add_cookie*)>
<!ATTLIST http
    contents          CDATA #IMPLIED
    contents_from_file CDATA #IMPLIED
    content_type       CDATA #IMPLIED
    if_modified_since  CDATA #IMPLIED
    method             (GET | POST | PUT | PATCH | DELETE | HEAD | PROPFIND |
↳PROPPATCH | COPY | MOVE | LOCK | UNLOCK | MKCOL | MKACTIVITY | OPTIONS | REPORT |
↳VERSION-CONTROL | MERGE | CHECKOUT) "GET"
    url               CDATA #REQUIRED
    version           (1.0 | 1.1) "1.1" >

<!ELEMENT soap EMPTY >
<!ATTLIST soap action CDATA #REQUIRED >

<!ELEMENT dyn_variable EMPTY >
<!ATTLIST dyn_variable
    name          CDATA #REQUIRED
    xpath         CDATA #IMPLIED
    re            CDATA #IMPLIED
    jsonpath      CDATA #IMPLIED
    pgsql_expr    CDATA #IMPLIED
    regexp        CDATA #IMPLIED
    header        CDATA #IMPLIED
    decode        (html_entities | false) "false" >

<!ELEMENT http_header EMPTY >
<!ATTLIST http_header
    name          CDATA #REQUIRED
    encoding      CDATA #IMPLIED
    value         CDATA #IMPLIED >

<!ELEMENT add_cookie EMPTY >
<!ATTLIST add_cookie
    key          CDATA #REQUIRED
    domain       CDATA #IMPLIED
    path         CDATA #IMPLIED
    value        CDATA #REQUIRED >

<!ELEMENT www_authenticate EMPTY >
<!ATTLIST www_authenticate
    passwd CDATA #REQUIRED
    userid CDATA #REQUIRED
    nonce  CDATA #IMPLIED
    opaque CDATA #IMPLIED
    cnonce CDATA #IMPLIED
    nc     CDATA #IMPLIED
    realm  CDATA #IMPLIED
    qop    CDATA #IMPLIED
    type   (basic | digest) "basic" >

<!ELEMENT oauth EMPTY >
<!ATTLIST oauth

```

```

    consumer_key CDATA #REQUIRED
    consumer_secret CDATA #REQUIRED
    access_token CDATA #IMPLIED
    access_token_secret CDATA #IMPLIED
    method (HMAC-SHA1 | PLAINTEXT | RSA-SHA1) "HMAC-SHA1">

<!--ELEMENT jabber (xmpp_authenticate?) -->
<!--ATTLIST jabber
    ack (global | local | no_ack | parse | bidi_ack) #REQUIRED
    destination (online | offline | random | unique | previous) "random"
    id NMTOKEN #IMPLIED
    size NMTOKEN "0"
    data CDATA #IMPLIED
    type NMTOKEN #REQUIRED
    stamped (true | false) "false"
    show (away|chat|dnd|xa) "chat"
    status CDATA "Available"
    nick CDATA #IMPLIED
    room CDATA #IMPLIED
    group CDATA #IMPLIED
    node CDATA #IMPLIED
    send_version (true | false) "false"
    regexp CDATA #IMPLIED
    resource CDATA "tsung"
    node_type CDATA #IMPLIED
    version CDATA #IMPLIED
    cacertfile CDATA #IMPLIED
    keyfile CDATA #IMPLIED
    keypass CDATA #IMPLIED
    certfile CDATA #IMPLIED
    subid CDATA #IMPLIED >

<!--ELEMENT xmpp_authenticate EMPTY -->
<!--ATTLIST xmpp_authenticate
    passwd CDATA #REQUIRED
    username CDATA #REQUIRED >

<!--ELEMENT fs EMPTY -->
<!--ATTLIST fs
    cmd (read|write|open|delete|stat|copy|read_chunk|write_chunk|close|make_dir|del_
    ↪dir|make_symlink) "write"
    path CDATA #IMPLIED
    size CDATA "1024"
    position CDATA #IMPLIED
    mode (read | write | append ) #IMPLIED
    dest CDATA #IMPLIED
>

<!--ELEMENT shell EMPTY -->
<!--ATTLIST shell
    cmd CDATA #REQUIRED
    args CDATA ""
>

<!--ELEMENT job EMPTY -->
<!--ATTLIST job

```

```

type      (oar|torque) "oar"
req       (submit|delete|stat|suspend|resume|wait__jobs) #REQUIRED
script    CDATA      #IMPLIED
walltime  CDATA      #IMPLIED
duration  CDATA      #IMPLIED
jobid     CDATA      #IMPLIED
resources CDATA      #IMPLIED
nodes     CDATA      #IMPLIED
queue     CDATA      #IMPLIED
options   CDATA      #IMPLIED
user      CDATA      #IMPLIED
name      CDATA      "tsung"
notify_port CDATA    #IMPLIED
notify_script CDATA  #IMPLIED
>

<!ELEMENT pgsql (#PCDATA) >
<!--ATTLIST pgsql
  password    CDATA      #IMPLIED
  database     CDATA      #IMPLIED
  username     CDATA      #IMPLIED
  name_portal  CDATA      #IMPLIED
  name_prepared CDATA      #IMPLIED
  query        CDATA      #IMPLIED
  parameters   CDATA      #IMPLIED
  max_rows     CDATA      "0"
  formats       CDATA      #IMPLIED
  formats_results CDATA    #IMPLIED
  contents_from_file CDATA #IMPLIED
  type         (connect | authenticate | sql | close | bind | parse | cancel|call|_
→sync | execute | describe | flush | copy | copydone| copyfail) #REQUIRED >

<!ELEMENT mysql (#PCDATA) >
<!--ATTLIST mysql
  password    CDATA      #IMPLIED
  database     CDATA      #IMPLIED
  username     CDATA      #IMPLIED
  type         (connect | authenticate | sql | close) #REQUIRED >

<!ELEMENT raw EMPTY >
<!--ATTLIST raw
  ack          (global | local | no_ack) #REQUIRED
  datasize    CDATA      #IMPLIED
  data        CDATA      #IMPLIED>

<!ELEMENT ldap (attr* | modification*) >
<!--ATTLIST ldap
  password    CDATA      #IMPLIED
  user        CDATA      #IMPLIED
  type        (bind | unbind | search | start_tls | add | modify ) #REQUIRED
  result_var  CDATA      #IMPLIED
  filter      CDATA      #IMPLIED
  base        CDATA      #IMPLIED
  scope       (singleLevel | baseObject | wholeSubtree) #IMPLIED
  cacertfile  CDATA      #IMPLIED
  keyfile     CDATA      #IMPLIED
  certfile    CDATA      #IMPLIED
  dn          CDATA      #IMPLIED

```

```
>
<!--ELEMENT websocket (#PCDATA) -->
<!--ATTLIST websocket
    type (connect | message | close) #REQUIRED
    ack (no_ack | parse) #IMPLIED
    frame (binary | text) #IMPLIED
    origin CDATA ""
    subprotocols CDATA ""
    path CDATA "/" -->

<!--ELEMENT amqp (#PCDATA) -->
<!--ATTLIST amqp
    type CDATA #REQUIRED
    vhost CDATA "/"
    channel CDATA "-1"
    exchange CDATA ""
    routing_key CDATA ""
    payload CDATA ""
    payload_size CDATA "100"
    prefetch_size CDATA "0"
    prefetch_count CDATA "0"
    persistent CDATA "false"
    queue CDATA ""
    timeout CDATA "1"
    ack CDATA "false" -->

<!--ELEMENT mqtt (#PCDATA) -->
<!--ATTLIST mqtt
    type CDATA #REQUIRED
    clean_start CDATA "false"
    keepalive CDATA "10"
    will_topic CDATA ""
    will_qos CDATA "0"
    will_msg CDATA ""
    will_retain CDATA "false"
    topic CDATA ""
    qos CDATA "0"
    retained CDATA "false"
    timeout CDATA "1"
    username CDATA ""
    password CDATA ""-->

<!--ELEMENT modification (attr*) -->
<!--ATTLIST modification
    type CDATA #REQUIRED-->

<!--ELEMENT attr (value+) -->
<!--ATTLIST attr
    type CDATA #REQUIRED-->

<!--ELEMENT value (#PCDATA) -->

<!--ELEMENT setdynvars (var*) -->
<!--ATTLIST setdynvars
    sourcetype (random_string | urandom_string | random_number |
        file | erlang | eval | jsonpath | value | server) #REQUIRED
```

```

    callback      CDATA      #IMPLIED
    code          CDATA      #IMPLIED
    fileid        CDATA      #IMPLIED
    order         (iter | random ) #IMPLIED
    delimiter     CDATA      #IMPLIED
    length        CDATA      #IMPLIED
    start         CDATA      #IMPLIED
    end           CDATA      #IMPLIED
    from          CDATA      #IMPLIED
    jsonpath      CDATA      #IMPLIED
    value         CDATA      #IMPLIED
  >
<!--ELEMENT var (#PCDATA) -->
<!--ATTLIST var
    name CDATA #REQUIRED-->

<!--ELEMENT for (request | thinktime | transaction | setdynvars | for |
    if | repeat | change_type | foreach | interaction | abort )+-->
<!--ATTLIST for
    var      CDATA      #REQUIRED
    from     CDATA      #REQUIRED
    to       CDATA      #REQUIRED
    incr     NMTOKEN    "1">

<!--ELEMENT foreach (request | thinktime | transaction | setdynvars | foreach |
    if | repeat | change_type | for | interaction | abort )+-->
<!--ATTLIST foreach
    name     NMTOKEN    #REQUIRED
    in       NMTOKEN    #REQUIRED
    include  CDATA      #IMPLIED
    exclude  CDATA      #IMPLIED
  >

<!--ELEMENT repeat (request | thinktime | transaction | setdynvars | for | repeat
| while | if | until | change_type | foreach | interaction | abort )+-->
<!--ATTLIST repeat
    name     NMTOKEN    #REQUIRED
    max_repeat NMTOKEN    "20">

<!--ELEMENT if (request | thinktime | transaction | setdynvars | for | repeat
| while | if | until | change_type | foreach | interaction | abort )+-->
<!--ATTLIST if
    var      CDATA #REQUIRED
    eq       CDATA #IMPLIED
    neq      CDATA #IMPLIED
    gt       CDATA #IMPLIED
    gte      CDATA #IMPLIED
    lt       CDATA #IMPLIED
    lte      CDATA #IMPLIED >

<!--ELEMENT while EMPTY-->
<!--ATTLIST while
    var      CDATA #REQUIRED
    eq       CDATA #IMPLIED
    neq      CDATA #IMPLIED
    gt       CDATA #IMPLIED
    gte      CDATA #IMPLIED
    lt       CDATA #IMPLIED

```

```
    lte          CDATA #IMPLIED >

<!ELEMENT until EMPTY>
<!ATTLIST until
    var          CDATA #REQUIRED
    eq           CDATA #IMPLIED
    neq          CDATA #IMPLIED
    gt           CDATA #IMPLIED
    gte          CDATA #IMPLIED
    lt           CDATA #IMPLIED
    lte          CDATA #IMPLIED >
```

CHAPTER 14

Indices and tables

- `genindex`
- `search`

A

abort, 60
apply_to_content, 59
arrivalphase, 26
arrivalrate, 27

B

batch, 24

C

callback, 57
change_type, 51
client, 22
cpu, 23

D

delimiter, 57
direct ip, 24
dtd, 85
dumptraffic, 21
duration, 28
dyn_variable, 54

E

emfile, 76
encoding, 21

F

faq, 73
fileid, 57
for, 59
foreach, 61

G

global_ack_timeout, 29
global_number, 15

H

hibernate, 30

host, 22
http_use_server_as_proxy, 14

I

idle_timeout, 29
if, 60
interarrival, 15, 27
ip, 23
iprange, 24
iter, 57

J

jabber, 35
json, 63
jsonpath, 56

L

load, 26
loglevel, 22

M

match, 58
max_ssh_startup, 28
maxnumber, 15, 27
maxusers, 24
munin, 26

O

options, 28
override, 28

P

page, 64
presence, 15
proxy, 17

R

rate_limit, 30
record_tag, 19

- redirect, [77](#)
- repeat, [60](#)
- RFC
 - RFC 3253, [6](#)
 - RFC 3921, [39](#)
 - RFC 4918, [6](#)
 - RFC 6455, [7](#)

S

- sample, [65](#)
- sample_counter, [65](#)
- sasl plain, [38](#)
- scan_intf, [24](#)
- scheduler, [24](#)
- seed, [31](#)
- server, [22](#)
- session, [32](#)
- setdynvars, [57](#)
- skip_headers, [59](#)
- snmp, [25](#)
- ssl_ciphers, [28](#)
- ssl_reuse_sessions, [28](#)
- ssl_versions, [28](#)
- start_time, [27](#)

T

- tag, [61](#)
- tcp_rcv_buffer, [28](#)
- tcp_snd_buffer, [28](#)
- thinktime, [28](#)
- thinktimes, [32](#)
- ts_http, [14](#)
- ts_jabber, [15](#)
- ts_user_server, [53](#)
- tsung-recorder, [17](#)

U

- udp_rcv_buffer, [28](#)
- udp_snd_buffer, [28](#)
- until, [60](#)
- use_controller_vm, [22](#)

W

- weight, [23](#)
- while, [60](#)

X

- xpath, [55](#)