March 3rd 2020

CLIFv2.3 user manual

http://clif.ow2.org/

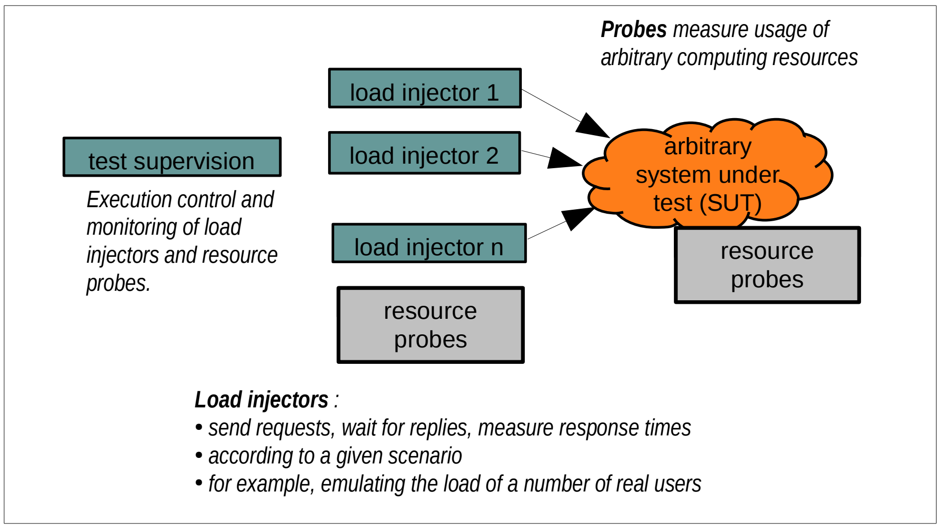
Table of contents

1. Introduction

CLIF is a component-oriented software framework written in Java, designed for load testing purposes of any kind of target system. By load testing, we mean generating traffic on a System Under Test in order to measure its performance, typically in terms of request response time or throughput, and assess its scalability and limits, while observing the computing resources usage.

Basically, CLIF offers the following features:

- deployment, remote control and monitoring of distributed load injectors;
- deployment, remote control and monitoring of distributed probes;
- final collection of measurements produced by these distributed probes and load injectors.


Probes *measure usage of arbitrary computing resources*
load injector 1
load injector 2
load injector n
test supervision
Execution control and monitoring of load injectors and resource probes.
arbitrary system under test (SUT)
resource probes
resource probes
Load injectors *:*
• send requests, wait for replies, measure response times
• according to a given scenario
• for example, emulating the load of a number of real users

Analysis tools for these measurements will be provided as soon as possible. For the time being, all measurements are available as CSV (comma separated values)-formated text files.

Thanks to its component-based framework approach, CLIF is easily customizable and extensible to particular needs, for example, in terms of specific injectors and probes, definition of load generation scenarios, storage of measurements, user (tester) skills, integration to a test management platform, etc. For instance, user interfaces are available as command-line tools, Java Swing-based GUI and Eclipse-based GUI.

See installation manual for CLIF installation.

2. Key concepts

- *blade*
  an active component that can be deployed within a CLIF application, under control of the supervisor component, that provides statistical information about its execution (for monitoring purpose), and produce results stored by the storage component. Blades exist either as load injectors or probes.
- *CLIF application*
  set of deployed components making it possible to run a test. A CLIF application is a distributed component holding as sub-components: one supervisor, one storage, and an arbitrary number of probes and load injectors (aka blades).
- *CLIF server*
  a JVM with a bootstrap component that will locally handle blade deployment requests from the supervisor. In other words, one must run a CLIF server on a given computer in order to be able to deploy load injectors and probes. CLIF server have a name. They register themselves in the Registry with this name in order to be found by the deployment process.
- *code server*
  the code server is responsible for delivering Java byte-code and resource files on demand during the deployment process. This is achieved through a socket server with a specific protocol. As of current version, files greater than 2GB cannot be transfered.
  Common CLIF usages create a dedicated code server for each test deployment, However, it is also possible to share a common code server in order to support several test deployments in parallel.
- *collect, collection*
  action of getting all measurements, possibly disseminated through the blades by the storage proxy feature, into the storage component. Collection should not occur before a test is terminated.
- *deployment*
  local or remote instantiation of load injectors and probes (aka blades). During this process, Java byte-code and resource files may be loaded from the code server, through the network, and to the target JVM of the blade being deployed.
- *load injector*
  a component that conforms to the blade component type, whose activity consists in generating traffic on an arbitrary SUT, using arbitrary protocols, according to an arbitrary scenario.
- *probe*
  a component that conforms to the blade component type, whose activity consists in measuring the usage of an arbitrary computing resource. Probes may be deployed at the SUT's side, in order to better analyze and understand its performance, as well as at the load injectors' side, to check that they are performing all right (since saturating injectors may result in unreliable measurements or violated load scenarios).
- *(load) scenario*
  optional concept referring to the way a single load injector generates traffic, for instance by emulating the load of a variable number of users performing a variety of requests on the SUT. In other words, a scenario defines both shape and content of the traffic generated by a load injector.

- *Storage*
  centralized component for storing measurements produced by load injectors and probes (aka blades). The storage component is typically associated to a storage proxy feature supported by each blade.
- *Storage proxy*
  local buffering of measurements feature provided by blades in order to avoid flooding the network and the storage component, which could also disturb the test and spoil measurements.
- *Supervisor or supervision console*
  component responsible for controlling and monitoring of a test execution.
- *System under test (SUT)*
  an arbitrary system one wants to assess the performance of. It is typically composed of one or several computers, networks, etc. It has to be reachable, either directly or indirectly via some gateway, native library or any wrapping mechanism, from the Java Virtual Machine where CLIF servers are running.
- *Registry*
  a distributed naming service used by the deployment process to lookup CLIF servers and deploy load injectors and probes.
- *Test (execution)*
  execution (shot) of an already deployed test plan. A test ends under 3 possible conditions: completed, manually stopped or self-aborted.
- *Test plan*
  specifies a set of distributed load injectors and probes, including their instantiation arguments and the name of the CLIF servers where they must be deployed.

3. Registry and CLIF servers

3.1. Rationale

CLIF servers are necessary to deploy any test plan, since they host load injectors and probes. A CLIF server is identified by a name, which is registered in a Registry. In order to run, CLIF servers must be able to find this Registry, which implies:

1. that the Registry must be running before a CLIF server can be launched;
2. that parameters must be given to tell CLIF servers where to find the Registry.

3.2. Running a Registry

There are three ways of starting a Registry: running the Java Swing console GUI (section 7), using the Eclipse-based console GUI (section 6), or using the appropriate command (see subsection 8.2.2 or 8.3.2).

3.3. Configuring and running a CLIF server

A CLIF server requires a CLIF runtime environment, providing the CLIF command-line interface, such as the CLIF server or the CLIF Swing console. Refer to commands described by subsections 8.2.1/8.3.1 and 8.2.3/8.3.3 for details about configuring CLIF and running a CLIF server.

For more information, refer to the appendix on System properties in Appendix A page 43.

4. Probes

4.1. Rationale

When load testing, it is often a good idea to check the usage of computing resources, both at the SUT side and the injectors' side. For instance, one may imagine system probes measuring CPU usage percentage, memory consumption, network bandwidth, etc. But other probes may be imagined that measure the size of a request queue length, a cache usage, or any activity data of any kind of middleware/software element involved in the SUT.

With CLIF, you may include probes in a test plan, as a complement to load injectors. Probes are supposed to have their own activity, typically (but not necessarily) consisting in polling a resource to measure its usage. All measurements are available from the Storage component once the test execution is over and the collection process has completed, while statistical values may be retrieved by the supervision console for monitoring purpose during test execution, directly from the probe. These statistical values are moving statistics computing on the period between two consecutive retrievals.

4.2. Available probes

Probes delivered with CLIF all consist in a periodic measure of the resource. They all take two arguments that must be specified in the test plan: the polling period (in milliseconds) and the execution duration (in seconds). Although probes start measuring once initialized for convenience, this execution time is counted once actually running (i.e. started and not suspended). When terminated, no measure is performed anymore.

To set a probe in a test plan:

- enter its family name as the “class name” information field;
- select the “probe” type;
- select the CLIF server where to deploy this probe, making sure that the target CLIF server actually runs on a computing environment (hardware, operating system or whatever) that is compatible with the probe family (see table below);
- enter the specific argument line, as explained hereafter.

Probes are specifically implemented for a variety of operating systems and architectures:

- Linux,
- FreeBSD,
- Mac OSX®,
- Windows®,
- Solaris®,
- AIX®,
- HP-UX

4.2.1. cpu probe

family/class name	cpu
measurements	• %CPU • %CPU user • %CPU kernel
alarms	*none*
2 arguments	polling period (ms), execution duration (s) Example: `1000 60`

4.2.2. disk probe

family/class name	disk
measurements	• issued reads • read throughput (kBytes/s) • issued writes • write throughput (kBytes/s) • IO time (micros) • queue • free space (kBytes) • used space % • free files • used files %
alarms	*none*
3 arguments	polling period (ms), execution duration (s), partition mount point Examples: • `1000 60 /usr/local` (Linux/Unix) • `1000 60 C:\` (Windows) Refer to subsection 8.2.18 or 8.3.18 to get the list of available partitions' mount points on a particular host. This list is also returned by the error message when the given mount point is not correct.

4.2.3. memory probe

family/class name	memory
measurements	• % used ram • used ram (MB) • free ram (MB) • free swap (MB) • % used swap • used swap(MB)
alarms	*none*
2 arguments	polling period (ms), execution duration (s) Example: `1000 60`

4.2.4. network probe

family/class name	network
measurements	• receive throughput (bit/s) • packets received • transmit throughput (bit/s) • packets transmitted • receive errors • receive overruns • receive drops • transmit errors • transmit overruns • transmit drops • transmit collisions
alarms	*none*
3 arguments	polling period (ms), execution duration (s), network interface identifier At your own convenience, the network interface identifier may be one of the followings: • *the network interface name*. It is easy to know on Linux/Unix from the output of command `ifconfig`. It is not directly available on Windows, unless using the CLIF utility described at subsection 8.2.18 and 8.3.18. • *the network interface's IP address* - always easy to know on any system, for example, from the output of command `ifconfig` on Linux/Unix, or command `ipconfig` on Windows. The drawback is that this address may change from one system boot to another when the network interface configuration is not static (use of DHCP). • *the network interface description.* It is the same as the network interface name on Linux/Unix. On Windows, the description is given by the system command `ipconfig /all`, but take care to **discard all white space characters**. In any case, CLIF provides a utility to get the list of available network interfaces, with their name, IP address and description, on a particular host (refer to subsection 8.2.18 or 8.3.18). This list is also returned by the error message when the given network interface identifier is not correct. Examples: • `1000 60 eth0` (Linux) • `1000 60 192.168.1.1` • `1000 60 BroadcomNetXtreme57xxGigabitController` (Windows)

4.2.5. jvm probe

family/class name	jvm
measurements	• free memory (MB) • used memory % • free usable memory %
alarms	An alarm with severity level “Info” is generated on each JVM garbage collection.
2 arguments	polling period (ms), execution duration (s) Example: `1000 60`
notes	This probe must be deployed in the JVM to monitor, which involves installing and integrating a CLIF server to the JVM(s) of the system under test. In certain conditions, some garbage collection alarms may be missed. This probe is system-independent.

4.2.6. jmx_jvm probe

family/class name	jmx_jvm
measurements	• free memory (MB) • used memory % • free usable memory %
alarms	An alarm with severity level “Warning” is generated when the connection with the JMX agent of the target JVM could not be established.

3 arguments	polling period (ms), execution duration (s), configuration file name
	The configuration file must set these properties:
	`# username and password used for the JMX connection`
	`server.user.name =`
	`server.user.password =`
	`# connection parameters to the MBean server`
	`server.host.name =`
	`server.host.port =`
	`server.connection.protocol = rmi`
	`server.jmx.connection.jndiroot = /jndi/iiop://`
	`server.jmx.protocol_provider_package = com.sun.jmx.remote.protocol`
	`server.jmx.location =`
	`server.jmx.mbeanservername =`
	Refer to the documentation of your Java runtime to find out how to activate and contact its JMX agent.
	Example: `1000 60 jonasjvm.props`
	with the following content for file `jonasjvm.props`:
	`# WARNING: activate first jvm jmx connection by adding option`
	`# -Dcom.sun.management.jmxremote to JONAS_OPTS`
	`server.host.name=`*`target JVM host`*
	`server.host.port=1099`
	`server.connection.protocol=rmi`
	`server.jmx.connection.jndiroot=/jndi/rmi://`
	`server.jmx.protocol_provider_package=com.sun.jmx.remote.protocol`
	`server.jmx.location=`
	`server.jmx.mbeanservername=jrmpconnector_jonas`

notes	This probe is a “remote” probe in that it may be deployed anywhere, and remotely interrogates the target JVM. The advantage is that it does not require to install and integrate a CLIF runtime to the target system. The drawback is that it produces some network traffic during test executions (although it should be a small traffic). This probe is system-independent but is only able to monitor a Sun/Oracle JVM.

4.2.7. rtp probe

family/class name	rtp
measurements	• number of packets per second • cumulative number of packets lost • minimum time jitter (ms) • maximum time jitter (ms) • average time jitter (ms) • standard deviation of time jitter (ms) • number of jumps per second • number of inversions per second
alarms	none
arguments	polling period (ms), execution duration (s), port or port range to monitor Examples: • `1000 60 40000-40002` monitor ports 40000 to 40002 • `1000 60 40000` monitor port 40000 • `1000 60 40000-40004/2` monitor ports 40000, 40002, 40004
note	This probe is system-independent

5. Load injectors and ISAC

5.1. Rationale

Load injectors are set in a CLIF test plan in order to generate traffic on the SUT. With CLIF, you may use and imagine any kind of way to define and execute your load scenarios, on any kind of SUT. You may even mix a variety of load injectors in the same test plan. This is the reason why you must set a class name for each load injector you define in a test plan, and set an arbitrary line of arguments, specifically to the actual load injector you use. Fortunately for non-programmers, CLIF comes with the ISAC extension in order to provide an easy, powerful and user-friendly way to define load scenarios. Luckily for Java programmers, they may also define their own load injectors.

5.2. ISAC is a Scenario Architecture for CLIF

With ISAC, testers are given a way to define load scenarios by combining:

- definitions of elementary behaviors, typically representing users;
- optional definitions of load profiles setting the population (i.e. the number of active instances) of each behavior as a function of time.

5.2.1. behaviors

An ISAC behavior basically consists in a sequence of actions (requests) on the SUT interlaced with delays (think times). It may be enriched with the following constructs:

- conditional loop: `while` <condition>
- conditional branching: `if` <condition> `then` <true_branch> `else` <false_branch>
- probabilistic branching: `nchoice` `choice`<weight_1, branch_1> `choice`<weight_2, branch_2> ... `choice`<weight_n, branch_n>
  where weight_i is an integer representing the chance of executing branch_i (in other words, the probability of executing branch_i equals weight_i divided by ∑ weight_j)
- preemptive condition: `preemptive` <condition, branch>
  program <branch> will exit as soon as <condition> is false (this condition is actually evaluated before executing each instruction in the branch)

5.2.2. load profiles

Load profiles enables predefining how the population of each behavior will evolve, by setting the number of active instances according to time. A load profile is a sequence of lines or squares. For each load profile, a flag states if active instances shall be stopped to enforce a decrease of the population, or if the extra behaviors shall complete in a kind of a "lazy" approach.

5.2.3. ISAC plug-ins

A behavior can be understood as a logic definition, a kind of a skeleton. In order to actually generate traffic on the SUT, this skeleton must be associated to one or more ISAC plug-ins. Plug-ins are external Java libraries, that are responsible for:

- performing actions (i.e. generating requests) on the SUT, whose response times will be measured, using and managing specific protocols (e.g. HTTP, DNS, JDBC, TCP/IP, DHCP, SIP, LDAP or whatever);
- providing conditions used by the behaviors' conditional statements (if-then-else, while, preemptive);
- providing timers to implement delays (think time), for example with specific random distributions or computed in some arbitrary way;
- providing ad hoc controls for the plug-in itself (e.g. to change some settings);
- providing support for external data provisioning (e.g. a database of product references or a file containing identifier-password pairs for some user accounts), used as parameters by the behaviors.

5.2.4. Writing an ISAC scenario

ISAC scenarios are stored in and read from XML files, with extension ".xis" (standing for XML Isac Scenario). An ISAC scenario holds three main sections:

1. a section for plug-in imports, where default/initialization parameters can be set. A plug-in may be imported more than once if necessary: for each imported plug-in, each instance of each behavior will hold a sort of private context (called session object). Each imported plug-in is designated via a unique identifier.

2. a section for behaviors definition. All actions (aka samples), conditions (aka tests), controls and delays (aka timers) must refer to an imported plug-in using its identifier. For each call to the plug-in, specific parameter strings may be set. Those strings may hold variables: when the pattern `${plugin-identifier:key}` is found, it is replaced at runtime by a value that the designated plug-in associates with the provided key string. The designated plug-in must be a "data provider" type plug-in, and the interpretation of the key depends on it (refer to the documentation of the data provider plug-in).

3. an optional section for load profiles, with (at most) one profile per behavior.

The most user-friendly way to edit a scenario is to use the Eclipse-based ISAC graphical editor (see section 6). The alternative is to use an XML or text editor (the DTD of ISAC scenarios is given in appendix page 50).

5.2.5. Recording an ISAC scenario for HTTP

In order to make realistic scenarios corresponding to real users behaviors, web interactions (sessions) can be recorded in ISAC scenario. It consists in using a recording HTTP proxy called MaxQ, available from the download section of CLIF's forge (http://forge.ow2.org/projects/clif/), as well as from tigris.org open source community (http://maxq.tigris.org/). MasQ will generate an ISAC scenario with all performed HTTP requests, and possibly all think times elapsed between two consecutive requests.

This Java tool may be used either as a standalone tool, or through an **Eclipse wizard** embedded in CLIF's Eclipse-based console (see section 6).

To record an ISAC scenario with the standalone version of MaxQ:

1. You have to edit the maxq.properties file and to choose which timer will be used during the injection (ConstantTimer and RamdomTimer are available). You can also specify on which port starts MaxQ. By default, it starts on the port 8090.
2. You have to configure your web browser to go through a proxy for Http requests.
3. Then you have to click on "File" -> "New" -> "ISAC scenario". At this point, the proxy is started but doesn't record ISAC scenario yet: it works as a transparent proxy.
4. Click on "Test" -> "Start Recording". Now, all requests going from the web browser to a server will be stored in the ISAC scenario.
5. At the end of the web session, click on "Test" -> "Stop Recording". A pop-up appears to select a name and a destination to save the file. Give a name with the extension ".xis". Then save.

Now you have a scenario corresponding to a user behavior. You can import it in your Clif Console to edit the load profile in order to replay it on a large scale.

5.2.6. Deploying and executing an ISAC scenario

Remember that a scenario is local to each load injector. When editing your test plan, the key idea is to use the ISAC execution engine as a load injector, and to set the test plan file as argument:

- class name: `IsacRunner`
- arguments: `myScenario.xis`

Your code server path should include the directory where your scenario file is, in order to benefit from the automatic remote loading of the scenario file by every remote ISAC execution engine you may have defined in your test plan.

A number of the execution engine's parameters may be modified, including at runtime:

- about the engine itself (size of the thread pool, polling period for load profile management, tolerance on deadlines - see appendix page 53);
- about the active scenario, in particular the number of active instances (population) of each behavior.

ISAC scenarios end on completion (load profiles time have elapsed), failure (abort), or manual stop. As soon as at least one behavior population has been manually set, or when no load profile is defined for any behavior, the scenario must be manually stopped.

5.3. Synchronization pseudo-injector

5.3.1. Rationale

A test plan may include several load injectors which require some global synchronization between each other. When using ISAC scenarios, for example, virtual users belonging to different scenarios, possibly distributed among different CLIF servers, may need to synchronize with each other. ISAC provides a synchronization plug-in dedicated to this kind of feature.

5.3.2. Usage

In order to support global, distributed synchronization, a test plan must include (at least) one synchronization pseudo-injector:

- class name: `Synchro`
- arguments: `synchronization_domain_name duration_in_seconds [`*`lock`*$_1$`=`*`n`*$_1$` ...]`

The first argument is a synchronization domain name. This name may be freely chosen, but it must be unique in your test plan. This name will be necessary for your scenarios (e.g. through the synchronization ISAC plug-in) to connect to the synchronization server embedded in the pseudo-injector.

The second argument is the execution duration, in seconds. This allows for an autonomous termination of the synchronization pseudo-injector, with the completed status. Note that, even when completion is reached, the synchronization features are still operational. In other words, it works still after the duration has elapsed.

Subsequent arguments are optional. They allow for declaring one or several predefined *rendez-vous*, specifying the minimum number of notifications that must be received by the named lock before releasing it. Refer to the synchronization tool documentation (e.g. the synchronization ISAC plug-in's help) for details about this feature.

5.3.3. Alarms

The Synchro pseudo-injector generates an alarm on the first notification of each lock. This alarm has the following fields:

- date: elapsed time since the test initialization (in milliseconds)
- severity level: 0 (INFO)
- message: first notification of lock *lock_name*

6. Eclipse-based graphical user interface

6.1. Introduction

CLIF comes with an Eclipse-based Graphical User Interface. This GUI has 4 functions:

- a CLIF console for test deployment, execution and monitoring, including a test plan editor;
- a graphical editor for ISAC scenarios;
- a programming environment for ISAC plug-ins;
- a reporting environment[1].

To install and run the Eclipse-based Graphical user interface, see the Install Manual.

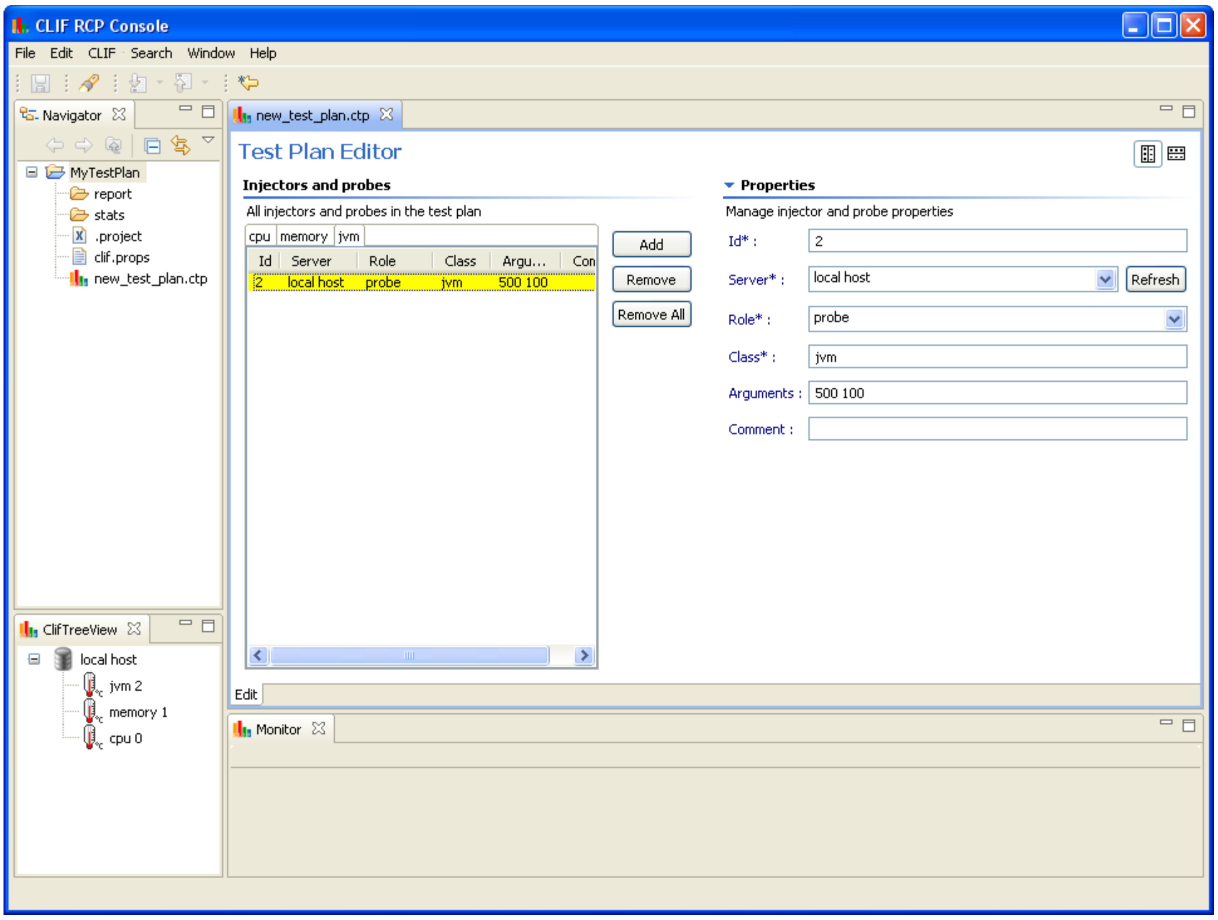
6.2. Run CLIF registry

Connection to the CLIF registry is updated on each test plan deployment or edition, according to the registry settings of the test plan's project (see the CLIF project settings in the Eclipse Preferences window, or the project's `clif.props` file). So you might run a standalone registry outside of the console, anywhere and rerun it anytime between two consecutive deployments.

Moreover, the console automatically runs a registry whenever it can't connect to the specified registry, using the specified registry port number. You can actually rely on this feature as a simple way to have a registry running without caring of starting it. The consequence is just that you have to run your CLIF servers after the console. This registry can't be stopped unless you quit the console, but still you can switch to any other registry by changing the settings.

1 currently as a preliminary version to be further completed by the CLIF team.

6.3. Test plan edition

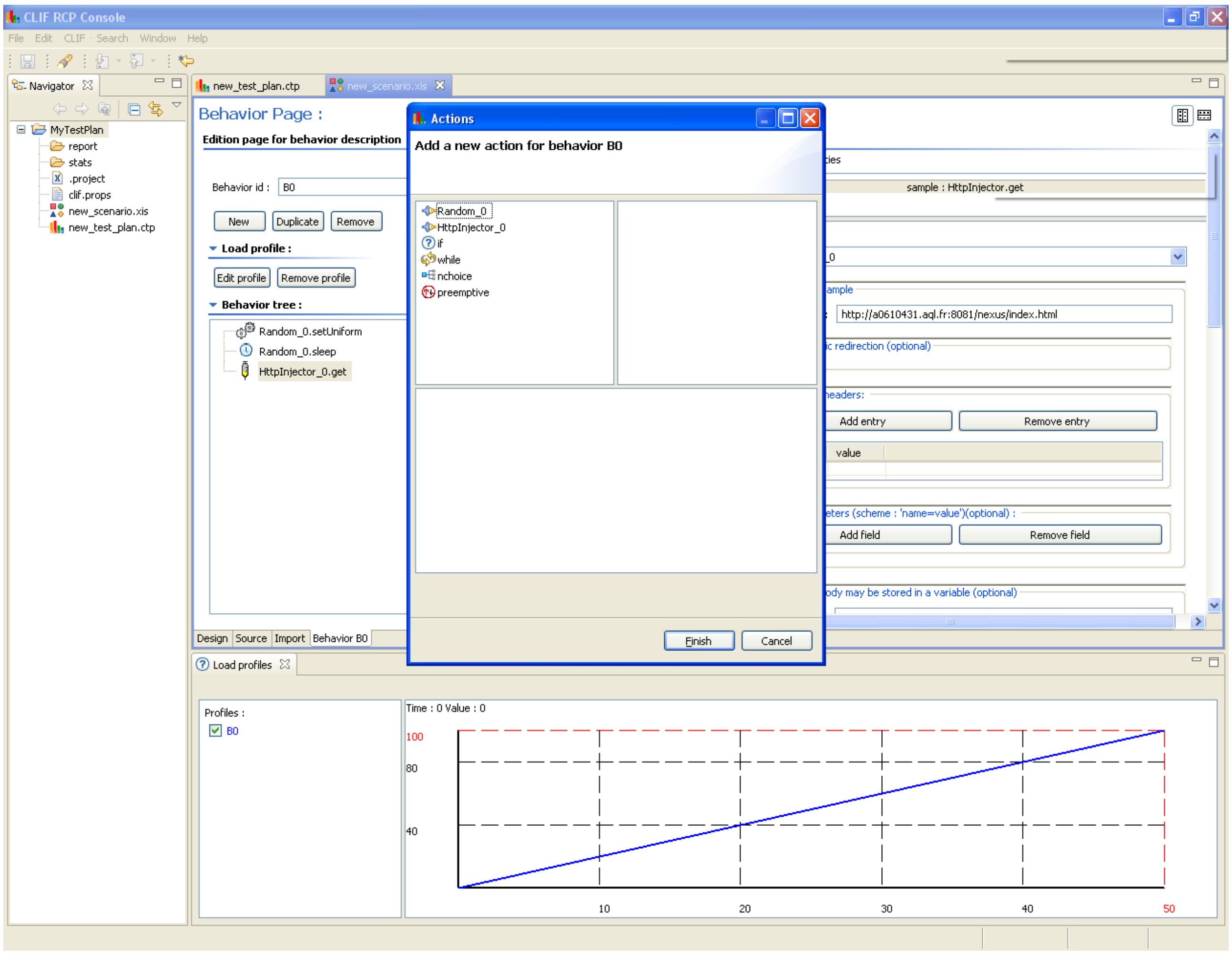


Id	Server	Role	Class	Argu...	Con
2	local host	probe	jvm	500 100	

Id* :	2
Server* :	local host
Role* :	probe
Class* :	jvm
Arguments :	500 100
Comment :	

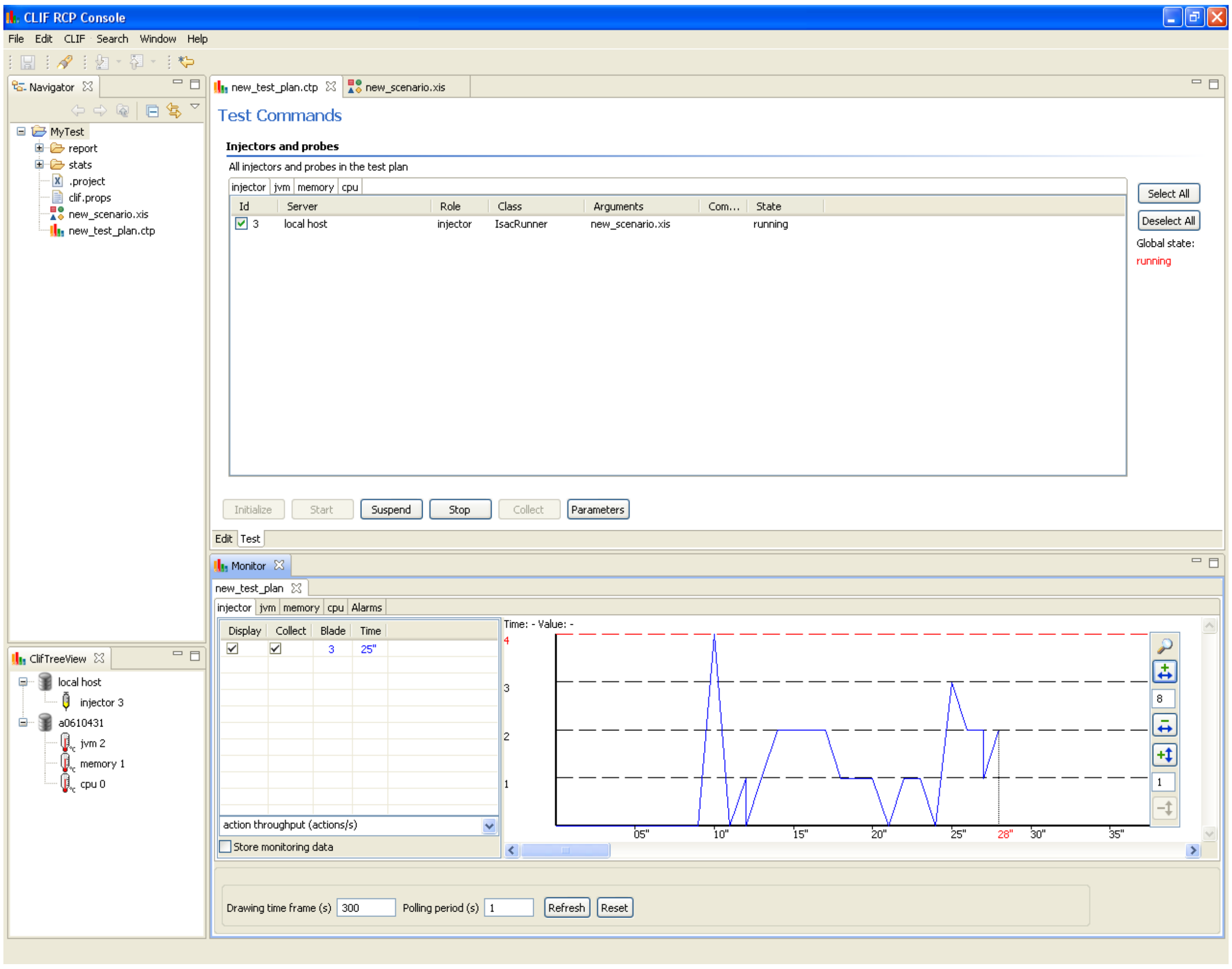
6.4. ISAC scenario edition

Please refer to the help section for the ISAC editor available from the Help menu. Refer also to section for information about ISAC.



6.5. test deployment and execution

Please refer to the help section available from the Help menu.



Test Commands

Injectors and probes

All injectors and probes in the test plan

injector | jvm | memory | cpu

Id	Server	Role	Class	Arguments	Com...	State
✓ 3	local host	injector	IsacRunner	new_scenario.xis		running

Global state: running

Display	Collect	Blade	Time
✓	✓	3	25"

action throughput (actions/s)

Store monitoring data

Time: - Value: -

Drawing time frame (s) 300 Polling period (s) 1

7. Java Swing-based graphical user interface

7.1. Introduction

CLIF comes with a Java/Swing-based Graphical User Interface. This GUI consists of a console for test deployment, execution and monitoring, including a test plan editor. It also provides an analysis tool to help produce test reports.

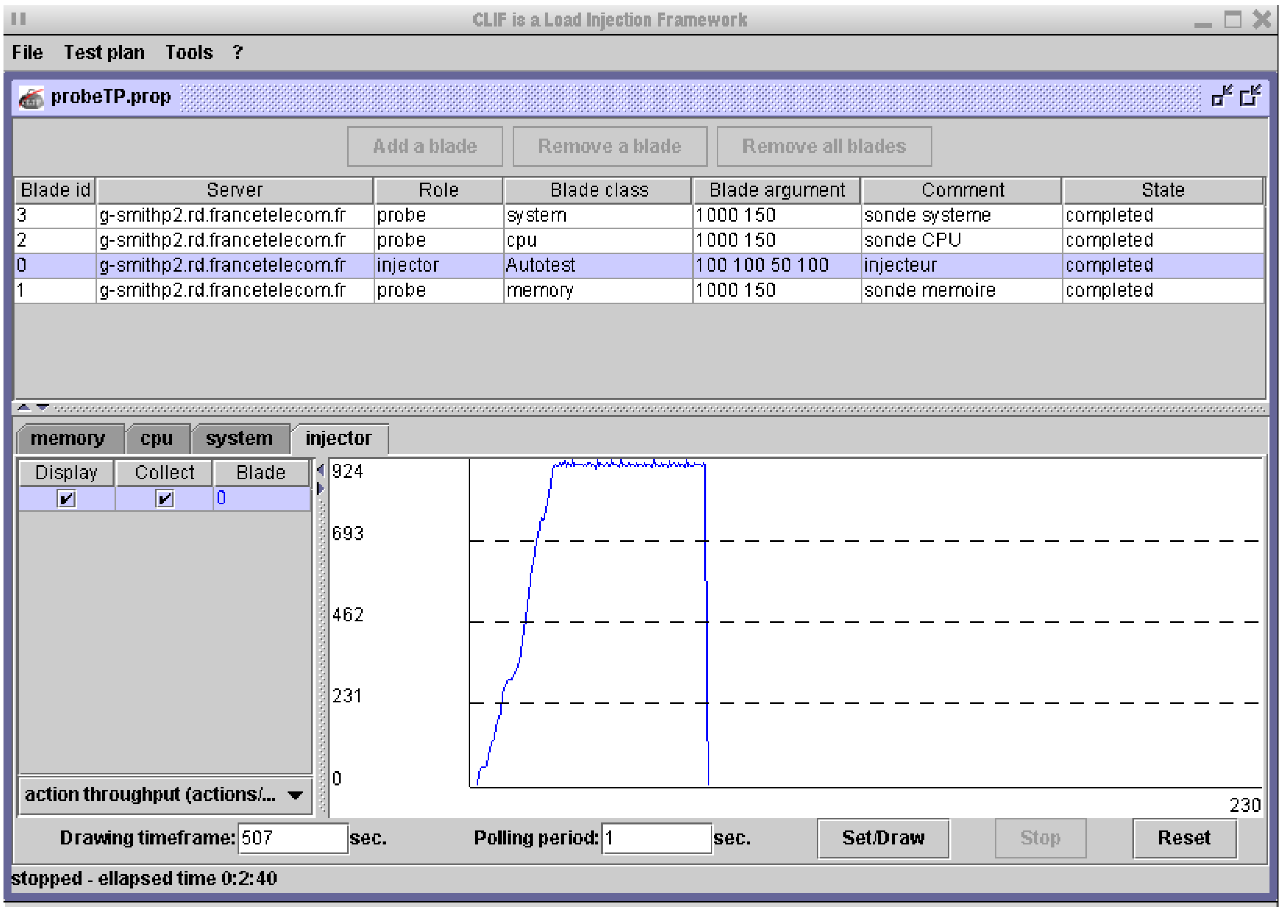
Compared to the Eclipse RCP-based console (see section 6), the Swing-based console has the advantage of light-weight, simplicity and operating-system independence. On the negative side, its simplicity springs from a reduced set of features. In particular, it does not provide an ISAC scenario editor nor an ISAC plug-ins creation wizard. As far as the test results analysis is concerned, the consoles provide different tools that suit different needs. The one provided by the Swing console is probably more straightforward to use, and rapidly gives graphical views, while the one provided by the Eclipse console is suited to the creation of long reports based on well-structured report templates. Of course, once a test has been run, any analysis tool may be used regardlessly of the user interface that has been used to run the test.

Note that the Swing console is actually embedded in the CLIF Eclipse-RCP distribution, since it provides the so-called CLIF runtime environment directory, located in the console plug-in path, i.e. something like `plugins/org.ow2.clif.console.plugin_x.x.x/`.

To install and run the Java Swing-based graphical user interface, see the Installation Manual.

7.2. Run CLIF registry

The GUI first tries to connect to a registry according to the registry configuration found in file etc/clif.props. If it can't connect, it creates a registry.



Blade id	Server	Role	Blade class	Blade argument	Comment	State
3	g-smithp2.rd.francetelecom.fr	probe	system	1000 150	sonde systeme	completed
2	g-smithp2.rd.francetelecom.fr	probe	cpu	1000 150	sonde CPU	completed
0	g-smithp2.rd.francetelecom.fr	injector	Autotest	100 100 50 100	injecteur	completed
1	g-smithp2.rd.francetelecom.fr	probe	memory	1000 150	sonde memoire	completed

Display	Collect	Blade
✓	✓	0

action throughput (actions/...

Drawing timeframe: 507 sec. Polling period: 1 sec.

stopped - ellapsed time 0:2:40

7.3. Test plan edition table

A test plan defines the probes and the injectors to be used, with their parameters, and where to deploy them. Remember that injectors and probes are uniformly designated as "blades". The table in the upper part is the test plan editor. Note that the bottom part (monitoring) is hidden as long as the test is not initialized. Note also that the test plan is not editable when the monitoring area is shown.

Each row of the test plan table defines a blade configuration, through 6 columns:

- **Blade id** is a unique identifier for the injector or probe to be deployed. A default id is automatically set when adding a new blade, but it may be freely changed by the user as long as it remains unique within current test plan;
- **Server** offers a choice between available CLIF servers, where the blade is to be deployed. The list of CLIF servers may be updated using option "Window > Refresh server list";
- **Role** specifies whether the blade is a probe or an injector;
- **Blade class** is where the user sets:
  - either the Java class to be instantiated as a load injector (fully qualified name, without trailing .class extension - see section 5),
  - or a family name in case of a probe (see section 4);
- **Blade argument** is an argument line that will be passed to the new blade instance at deployment time;
- **Comment** is an arbitrary user comment line.

The last column **State** is not editable. It shows state information about the blade (undeployed, deploying, deployed, starting, running, stopping, suspending, resuming, completed, aborted...).

Test plans may be saved and restored using options in the File menu.

7.4. Performance and resource usage monitoring

As soon as the test plan is deployed and initialized, the monitoring area pops up in the test plan window's bottom part. This area holds a set of tabbed panels:

- one for all injectors
- one for each probe family

For each panel, the user may set the monitoring time-frame, the polling period, and start or stop the monitoring process. Moreover, a check-box table at the left side of each panel makes it possible to selectively disable or enable the collect and display of monitoring data, for each blade.

7.5. File Menu

From this menu, the user can find options for saving and loading a test plan.

This menu also holds the "Quit" option to exit from CLIF console, which also terminates the registry where CLIF servers are registered. As a result, whenever you terminate a CLIF console, any remaining CLIF server will then become unreachable - you may stop these unreachable CLIF servers manually. Running the CLIF console again will create a new, empty registry, and then you may launch new CLIF servers. The user may not quit the console while a test is running (other wise, the behavior is undefined).

7.6. Test plan menu

This menu holds test deployment and control commands. There are 2 subsets of options:

- the first set holds test plan definition and deployment commands
  - option **Refresh server list** updates the list of available CLIF servers,
  - option **Edit** switches to test plan edition mode, when enabled (i.e. when not already in edition mode, and when no deployed test is currently running),
  - option **Deploy** deploys the probes and injectors defined by current test plan
- the second set holds test control commands
  - command **initialize** initializes all the blades so that they are actually ready to start;
  - commands **start**, **suspend**, **resume** and **stop** respectively start, suspend, resume and stop the execution of all blades;
  - command **collect** tells the storage system to collect all test data from the blades (the actual effect of this command fully depends on the Storage component). This option may be used only after a test run. Collecting more than once after a test run has no effect; collecting is not mandatory, which means that the user may not collect data if s/he is not interested in the test results.

7.7. Tools menu

This menu displays on/off additional tools:

7.7.1. Basic analyzer

The Basic analyzer tool provides a very simple analysis tool sample. It must be used once at least one test execution is complete, since it needs to get measures from one test execution.

7.7.2. Quick graphical analyzer

The Quick Gaphical Analyzer tool intends to provide a powerful and efficient tool to fulfill test analysis and reporting needs. It is also embedded in the Eclipse-based console. This tool is currently under development, but some basic features may be used already. Documentation will be detailed as the tool development is progressing.

Report principle

A report is a set of pages. Each page holds:

- a number of data sets, built from CLIF's measures collected after a test execution, according to a number of possible filters;
- for each dataset, an optional section of statistical values computed from one specific metric;
- for each statistics section, an optional drawing of the selected metric in a graph section;
- a single graph section, where possible dataset drawings are superimposed.

Datasets

Datasets

There are three types of dataset (see File menu).

The basic type is the *Simple Dataset*. It represents a set of measures of a given type produced by a given load injector or a probe, from a given test execution. For example: the alarm events from one JVM probe of one test execution, or the action events from one load injector of one test execution. A number of selection filters may be used to keep only the measures of interest. For example, keep only requests of a given type or with a response time less than a given threshold. Filtering on dates is also possible to restrict analysis to a sub-period of the test execution.

The *Multiple Dataset* offers an efficient way to select a given event type produced by several load injectors or probes with the same filters. It is equivalent to creating as many simple datasets as chosen injectors or probes, but in a single operation. Further optional statistical analysis and drawing will be also defined once for all injectors and probes. Typical examples: create a multiple dataset including every CPU probe of a given test execution, in order to get per-CPU analysis and drawing superimposed on the same graph, or a given CPU probe in several test executions to compare CPU usage.

The *Aggregate Dataset* enables to create a kind of simple dataset containing the full set of events of a given type coming from several load injectors or probes measures. Typical usage: when using several load injectors in a test, to get the global response time and throughput in the analysis and drawings.

7.8. Help menu

This menu holds a single "**About...**" option, which displays CLIF version and compilation information. This information is important to get and mention whenever you report a problem using CLIF.

8. Command line user interface

8.1. Introduction

8.1.1. Rationale

Once you have created a test plan file (either using the Eclipse-based or the Java Swing-based GUI, or editing a text file with the appropriate syntax), you may deploy and run tests through command lines. Prior to any test plan deployment, one CLIF Registry (aka Fractal Registry) must be running. It will be used by every command to register or lookup components of the deployed test plan, especially the CLIF servers.

Some of these commands apply either to every probe and injector of a deployed test plan, or to a subset of them. In the latter case, you must specify an extra argument to give the list of the target injectors and probes identifiers (so-called blade identifier, as defined in the test plan): `-Dblades.id=id1:id2:...idn`. Note that separately managing probes and injectors can become tricky in big test plans... A typical usage of CLIF may not need this feature, and you would only make use of the commands' default global scope.

Authorized commands depend on the state of the injectors and probes. Refer to the appendices of the Developer Manual for details about the blade life-cycle.

The typical sequence of basic commands is the following: `config – registry – server – launch`. In the case of a test plan using only the default CLIF server “local host”, only the `launch` command is necessary.

8.1.2. Choosing the right command line interface

There are two command line interfaces available:

- the shell script-based interface
- the ant-based interface

Both come with exactly the same features. As a major difference, the shell script-based interface handles one CLIF configuration file per test project, while the ant-based interface relies on a single configuration file per CLIF installation, which makes it harder to share a CLIF runtime environment among several test projects and users. Moreover, automatic network configuration is not reliable with the ant-based interface: in complex cases, you may edit the configuration file and enter the right network addresses. As a result, the shell script-based command line interface is recommended and the use of the ant-based interface is now discouraged.

8.2. Shell script-based command line interface

These full-fledged control scripts are available in the `bin` directory of the CLIF runtime environment:

- `clifcmd` is a Bash script for Linux, MacOSX and Unix environments
- `clifcmd.bat` is a batch script for Windows environments

Their usage is identical for both: `clifcmd[.bat]` *command* `[arguments]`. The list of available *commands* is given in the following sub-sections (arguments between [ ] are optional).

8.2.1. Configure the CLIF environment

```
config [registry_host[:registry_port] [codeserver_host[:codeserver_port]]]
```

This command creates or updates `clif.opts` configuration file, in current directory, in order to set network-related properties:

- the IP address where the registry is to be made available (and possibly a custom port number, to override the default port number), so that CLIF servers can register;
- the IP address where the code server is to be made available (and possibly a custom port number, to override the default port number), so that necessary code and resources can be downloaded when deploying a test plan;
- in a transparent manner, this command also tries to connect to the registry and to run a smart selection process of network addresses to use for communication with CLIF servers. For example, non-routed IP addresses or translated addresses (Network Address Translation) are excluded. This selection is aborted if the registry is not running at the given (or default) address.

When run with no argument, configuration will default to `localhost` IP address and default port numbers for both the registry and the code server (respectively 1234 and 1357).

The first optional argument sets the IP address of the host where the CLIF registry is running or going to be launched. Optionally, the registry's default port number (1234) may be overridden to a custom port number.

The second optional argument sets the IP address of the host where the code server will be launched for upcoming test plan deployments. Optionally, the code server's default port number (1357) may be overridden to a custom port number. When this second argument is omitted, the IP address of the code server defaults to the registry's IP address.

You shall run this command first before running the registry, so that you can set the IP address, and possibly the custom port number, where the registry will be reachable. Then, you shall run the registry and run this command for all CLIF servers before running them, to configure their network properties and detect possible network issues.

Note. This command may take some time to complete: expect about 5 seconds per network interface.

8.2.2. Run the CLIF registry

```
registry
```

Runs the CLIF registry (aka Fractal Registry) so that CLIF servers can register. The registry will bind to the IP address and port number set during the configuration step. This command never returns unless the Registry process is killed.

When the shared code server mode is enabled, this command also starts a code server, that will be used for all further deployments.

Note. Only one Registry shall be launched on a given host (further attempts will just fail), unless you install another CLIF runtime environment and configure it with a different registry port.

8.2.3. Run a CLIF server

```
server [name]
```

Runs a CLIF server and registers it in the CLIF registry, either using current host name as server name, or using the provided argument as name. In either way, CLIF servers' names must match servers' names in the test plans to be deployed. Names are case-sensitive. A CLIF server can not be created without a reachable Registry.

This command never returns unless the CLIF server process is killed.

Notes:

- when a CLIF server is killed, its name remains registered until the registry is restarted, or another CLIF server is started and registered with the same server name. Then, a test plan deployment may fail not only because a CLIF server name is missing in the registry, but also because the corresponding CLIF server is no longer alive.
- whenever the Registry is restarted, all registered CLIF servers must be restarted also in order to be reachable again.
- the default CLIF server, named “local host”, is created by the deployment command and registered in the registry.

8.2.4. Print names of registered CLIF servers

```
listservers
```

Prints the names of CLIF servers currently registered in the registry.

Note. This command does not check for real availability of CLIF servers. A CLIF server may have registered some time ago and then it may have become unreachable for some reason, while this command will still list its name however.

8.2.5. Wait for registry and CLIF servers

```
waitservers [testplan.ctp]
```

Waits until the registry and, optionally, all CLIF servers used by a given test plan file are ready. Note the default CLIF server named "local host" is not taken into account, since it is created by the deploy command. This command is typically called before the `deploy` or `launch` command, to avoid a deployment failure due to missing CLIF servers.

This command output lists CLIF servers' names as soon as they are registered. If a CLIF server is missing in the output, and that the command does not exit, then either this CLIF server was not successfully started, or it has erroneously registered in another registry. In both cases, check the configuration of the missing CLIF server and (re)start it.

Note. This command only checks that some CLIF servers have registered with the required names. It does not check for real availability of CLIF servers: a CLIF server may have registered some time ago and then it may have become unreachable for some reason, while this command will still consider it as ready.

8.2.6. Deploy a test plan

```
deploy testplan_name testplan_file
```

Deploys a new test plan (probes and injectors) as defined by the given test plan file. The first argument sets a name to the deployed test plan. This name is then required by almost all other commands to control this deployed test plan. The second argument gives the path to the test plan definition file (usually ending with the `.ctp` extension, although this is not mandatory nor assumed by CLIF).

When the test plan to deploy involves CLIF servers other than the default "local host" CLIF server, then the registry and CLIF servers must be running prior to running this command. Otherwise, this command will handle the registry creation if necessary. In any case, this command creates the "local host" default CLIF server.

When successful, this command does not return, and should not be manually terminated as long as you want to use the deployed test plan.

Note. This command includes the launch of the CLIF code server that will deliver all necessary code and resources used by injectors and probes (such as workload scenario files or probe configuration files). Thus, the code server network configuration, among all involved CLIF servers, must be consistent with the network location of this code server.

8.2.7. Initialize a new test

```
init testplan_name testrun_id
```

Initializes all probes and injectors in a deployed test plan. The target deployed test plan is designated by its name (as set at deployment time). An identifier for this new test must be provided. This identifier will be further useful to the user to identify this test run, for instance when browsing measurements for analysis and reporting purpose.

8.2.8. Start a test

```
start testplan_name [id1:id2:...idN]
```

Starts probes and injectors of the given deployed test plan, or just a subset of them when specified. They must be initialized prior to this command.

8.2.9. Suspend a test

```
suspend testplan_name [id1:id2:...idN]
```

Suspends all probes and injectors of the given deployed test plan, or just a subset of them when specified. They must be running prior to this command.

8.2.10. Resume a test

```
resume testplan_name [id1:id2:...idN]
```

Resumes all probes and injectors of the given deployed test plan, or just a subset of them when specified. They must be suspended prior to this command.

8.2.11. Stop a test

```
stop testplan_name [id1:id2:...idN]
```

Definitively stops all probes and injectors of the given deployed test plan, or just a subset of them when specified. Stopping is possible for both running and suspended probes/injectors, as well as right after initialization.

Don't forget to use the `collect` command to gather all measurements to the host from where the test plan has been deployed, unless you are not interested in these measurements for some reason. Once a test is stopped, the same deployed test plan may be initialized again to run another test, and so on.

8.2.12. Wait for end of test

```
join testplan_name [id1:id2:...idN]
```

Waits until the probes and injectors of the given deployed test plan, or just a subset of them when specified, terminate.

8.2.13. Collect measurements

```
collect testplan_name [id1:id2:...idN]
```

Collects measurements generated by probes and injectors of the given deployed test plan, or just a subset of them when specified. Collecting is optional, i.e. the user may not collect results s/he is not interested in. This command waits for injectors and probes to be terminated prior to actually perform the collect operation.

8.2.14. Full run of a deployed test

```
run testplan_name testrun_id [id1:id2:...idN]
```

Automatic sequence of `init`, `start` and `collect` commands on all the probes and injectors of the given deployed test plan, or just a subset of them when specified (`init` is applied to all injectors and probes in any case).

8.2.15. Deployment and full test run

```
launch testplan_name testplan_file testrun_id
```

Automatic sequence of test plan deployment and then commands `init`, `start` and `collect` on all probes and injectors of the given test plan. This command exits when the full sequence is complete. As a major difference with the use of command `deploy`, which enables several consecutive runs on the same deployed test plan, the test plan is deployed and executed only once.

When the test plan to deploy involves CLIF servers other than the default "local host" CLIF server, then you shall run the registry and the necessary CLIF servers prior to invoking this command. Otherwise, there is no necessity for first running a registry.

8.2.16. Print injector or probe parameters

```
params testplan_name id
```

Lists all parameters of a probe or injector, designated by its identifier, among a deployed test plan. These parameters names and values are specific to the target probe or injector. Values may be changed using command `change` (see 8.2.17).

8.2.17. Change an injector or probe parameter value

```
change testplan_name id param_name param_value
```

Changes a parameter's value for a given injector or probe in a given deployed test plan.

8.2.18. Print probe help

```
probehelp probe_type
```

Prints a help message about the arguments that a probe of type *`probeType`* must be given when it is involved in a test plan. Existing probe types and their arguments are detailed in this documentation (see section 4.2), but this command is helpful for argument values that are not always straightforward to guess, like a network adapter name or a disk partition path.

8.2.19. Print response times statistics

```
quickstats [report_directory]
```

Prints a statistical synthesis for the latest test found in the default report directory, or in the provided report directory path. This synthesis aggregates all load injectors and gives, for each type of request as well as for all types of requests:

- the number of successful requests and errors,
- response times minimum, maximum, mean, median, standard deviation,
- the throughput.

Two system properties may be set in the CLIF configuration file to perform some cleaning of measurements, i.e. discard extreme values. See properties `clif.quickstats.limit` and `clif.quickstats.factor` in Appendix A p.43 for explanations.

8.2.20. Get CLIF version information

```
version
```

Prints version information about the operating system, the Java environment and CLIF. This information is mostly useful when interacting with CLIF support, to accurately and quickly identify the key elements of your CLIF environment.

8.2.21. Run graphical tools

```
gui
analyze
```

Run simple graphical user interfaces. These commands are available only in "swing" CLIF distributions (not in plain "server" distributions).

- `gui` runs a simplified CLIF console giving access to test plan edition, deployment and execution, as well as measurement analysis and test reporting;
- `analyze` runs just a subpart of the console, consisting of the main analysis and reporting graphical tool.

8.3. ant-based command-line interface

These commands are defined in the `build.xml` file available at CLIF runtime environment's root. They make use of a configuration file located in `etc/clif.props` in CLIF runtime environment. As a major difference with the shell script-based command line interface, the ant-based interface relies on a unique configuration file per CLIF runtime environment, instead of one per test project directory, making it harder to share a CLIF environment for different test projects and users.

The ant-based command line interface is deprecated and the use of the shell script-based command line described in previous subsection is recommended instead. Besides, the shell script-based command-line interface does not require the `ant` utility.

8.3.1. Configure the CLIF environment: config

```
ant [-Dregistry.host=myRegHost] [-Dregistry.port=1234]
[-Dcodeserver.host=myCodeServerHost] [-Dcodeserver.port=1357]
[-Dcodeserver.path=myCodesererPath] config
```

This command is a helper to update CLIF's configuration file located in `etc/clif.props`. The extension name of this configuration file is a bit misleading: although it actually defines Java properties, it also defines, more generally, JVM options, and it does not follow the common Java property file syntax. For this reason, it is recommended to use this helper command, which is sufficient for a common usage of CLIF. However, setting extra properties or adding custom JVM options will require to edit this file with a text editor. Refer to Appendix A page 43 to get the full list of CLIF customization properties.

This command is either interactive or non-interactive: when option `-Dregistry.host` is set in the command line, no user interaction is performed, while the user is prompted for a registry host value otherwise. In both cases, all other options are automatically set to their default values, which should be correct in common cases.

The CLIF configuration step must be performed on each CLIF runtime environment involved in a test plan deployment, before any other command. This includes the target CLIF servers, as well as the CLIF runtime from which you deploy test plans. When deploying with the Eclipse-based console, configuration is achieved through the GUI (see the CLIF properties of the test project).

Options:

`-Dregistry.host=...`: set the IP address or network name of the node where the registry is expected to be run. `localhost` is the initial value when no configuration is performed, but it only works when test plans are not be deployed over a network, but on the same node as the registry itself.

`-Dregistry.port=...`: set the TCP port number used by the CLIF registry. You may change the default value 1234 when it is not free or permitted on your registry node.

`-Dcodeserver.host=...`: set the IP address or network name of the node where the CLIF code server is expected to be run. The code server is automatically launched when you deploy a test plan, so that necessary resource files or even Java code can be transparently dispatched to all the CLIF servers involved in a distributed test plan. In common situations, you may run the test plan deployment on the same node as the registry. So, when this option is omitted, the `config` command assumes that the code server's address is the same as the registry's address. Then, you must set this option whenever you deploy a test plan from one node, while the registry is running on another node.

`-Dcodeserver.port=...`: set the TCP port number used by the code server. You may change the default value 1357 when it is not free or permitted on the node hosting the code server (the node you run the test plan deployment from).

`-Dcodeserver.path=...`: set the directories from where the code server should look for when CLIF servers ask for missing resource files or code when a test plan is being deployed. The default value is the current working directory (represented by “.”). This option is useful only in the CLIF runtime environment from which test plans are deployed; CLIF servers do not use this parameter.

Note. **The resulting network configuration may not always be correct**, especially in cases of multiple network interfaces, which may result in getting CLIF server, deployment or measurement collection problems (error or freeze). Then, you may have to edit the configuration file in order to fix some network settings.
This is the reason why it is recommended to switch to the shell script-based command line interface, for it generates correct network configurations thanks to a more advanced protocol (refer to 8.2).

8.3.2. Run CLIF Registry: registry

```
ant registry
```

Equivalent for command `clifcmd registry`. Refer to 8.2.2.

8.3.3. Run a CLIF server: server

```
ant [-Dserver.name=myClifServer1] server
```

Equivalent for command `clifcmd server [myClifServer1]`. Refer to 8.3.3.

8.3.4. Print the list of available CLIF servers: listservers

```
ant listservers
```

Equivalent for command `clifcmd listservers`. Refer to 8.2.4.

8.3.5. Wait for registry and CLIF servers: waitservers

```
ant -Dtestplan.file=myTestPlan.ctp waitservers
```

Equivalent for command `clifcmd waitservers myTestPlan.ctp`. Refer to 8.2.5.

8.3.6. Test plan deployment: deploy

```
ant -Dtestplan.name=name -Dtestplan.file=myTestPlan.ctp deploy
```

Equivalent for command `clifcmd deploy name myTestPlan.ctp`. Refer to 8.2.6.

8.3.7. Test initialization: init

```
ant -Dtestplan.name=name -Dtestrun.id=testId init
```

Equivalent for command `clifcmd init name testId`. Refer to 8.2.7.

8.3.8. Test execution start: start

```
ant -Dtestplan.name=name [-Dblades.id=id1:id2:...idn] start
```

Equivalent for command `clifcmd start name id1:id2:...idn`. Refer to 8.2.8.

8.3.9. Suspend test execution: suspend

```
ant -Dtestplan.name=name [-Dblades.id=id1:id2:...idn] suspend
```

Equivalent for command `clifcmd suspend name id1:id2:...idn`. Refer to 8.2.9.

8.3.10. Resume test execution: resume

```
ant -Dtestplan.name=name [-Dblades.id=id1:id2:...idn] resume
```

Equivalent for command `clifcmd resume name id1:id2:...idn`. Refer to 8.2.10.

8.3.11. Stop test execution: stop

```
ant -Dtestplan.name=name [-Dblades.id=id1:id2:...idn] stop
```

Equivalent for command `clifcmd stop name id1:id2:...idn`. Refer to 8.2.11.

8.3.12. Wait for a test execution to terminate: join

```
ant -Dtestplan.name=name [-Dblades.id=id1:id2:...idn] join
```

Equivalent for command `clifcmd join name id1:id2:...idn`. Refer to 8.2.12.

8.3.13. Collect test results (measurements): collect

```
ant -Dtestplan.name=name [-Dblades.id=id1:id2:...idn] collect
```

Equivalent for command `clifcmd collect name id1:id2:...idn`. Refer to 8.2.13.

8.3.14. Shortcut for full test execution process: run

```
ant -Dtestplan.name=name -Dtestrun.id=testId [-Dblades.id=id1:...idn] run
```

Equivalent for command `clifcmd run name testId id1:...idn`. Refer to 8.2.14.

8.3.15. Shortcut for full deployment and execution process: launch

```
ant   -Dtestplan.name=name   -Dtestrun.id=testId   -Dtestplan.file=myTestPlan.ctp
launch
```

Equivalent for command `clifcmd launch name myTestPlan.ctp testId`. Refer to 8.2.15.

8.3.16. Get specific runtime parameters of a probe or injector: params

```
ant -Dtestplan.name=name -Dblade.id=id params
```

Equivalent for command `clifcmd params name id`. Refer to 8.2.16.

8.3.17. Change a runtime parameter of a probe or injector: change

```
ant  -Dtestplan.name=name  -Dblade.id=id  -Dparam.name=param  -Dparam.value=value
change
```

Equivalent for command `clifcmd change name id param value`. Refer to 8.2.17.

8.3.18. Get help about a probe's arguments: probehelp

```
ant -Dprobe.type=probeType probehelp
```

Equivalent for command `clifcmd probehelp probeType`. Refer to 8.2.18.

8.3.19. Generate and print a quick statistical report: quickstats

```
ant        [-Dreport.dir=reportdirectory]        [-Dclif.quickstat.cleanfactor=f
-Dclif.quickstat.cleanlimit=p] quickstats
```

Equivalent for command `clifcmd quickstats reportdirectory`, except that optional parameters `f` (cleaning factor) and `p` (cleaning limit) may be given via the command line instead of just through the CLIF configuration file. Refer to 8.2.19.

8.3.20. Get CLIF version information

```
ant version
```

Equivalent for command `clifcmd version`. Refer to 8.2.20.

8.3.21. Run graphical tools

```
ant gui
```

```
ant analyze
```

Equivalent for commands `clifcmd gui` and `clifcmd analyze`. Refer to.

9. Using CLIF with Jenkins

9.1. What to do with it?

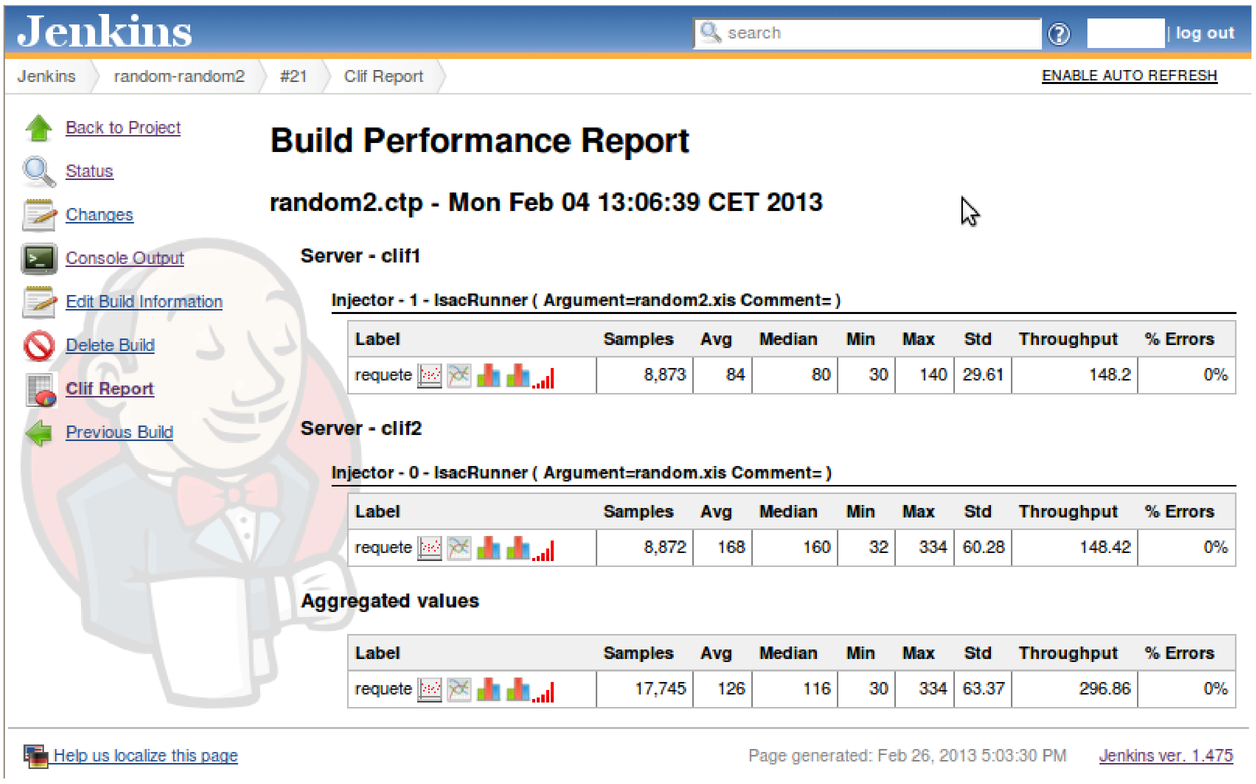
Jenkins CI is a popular continuous integration framework (http://jenkins-ci.org), dedicated to automating tests and reporting. CLIF provides a plug-in for Jenkins, available from the CLIF project's download area:

http://forge.ow2.org/project/download.php?group_id=57&file_id=20200

Thanks to Jenkins and the CLIF plug-in, you can:

- define performance test jobs directly from your CLIF test projects;
- use Jenkins as a web interface to deploy and run CLIF tests;
- get automatic performance reports, with both per-test detailed statistical analysis (including graphs) and performance trends graphs through consecutive tests;
- automate performance test runs, either in a continuous integration perspective, or to monitor performance and quality of experience of some running applications or services.

Please refer to the installation manual for setting your CLIF-enabled Jenkins environment.



Build Performance Report

random2.ctp - Mon Feb 04 13:06:39 CET 2013

Server - clif1

Injector - 1 - IsacRunner (Argument=random2.xis Comment=)

Label	Samples	Avg	Median	Min	Max	Std	Throughput	% Errors
requete	8,873	84	80	30	140	29.61	148.2	0%

Server - clif2

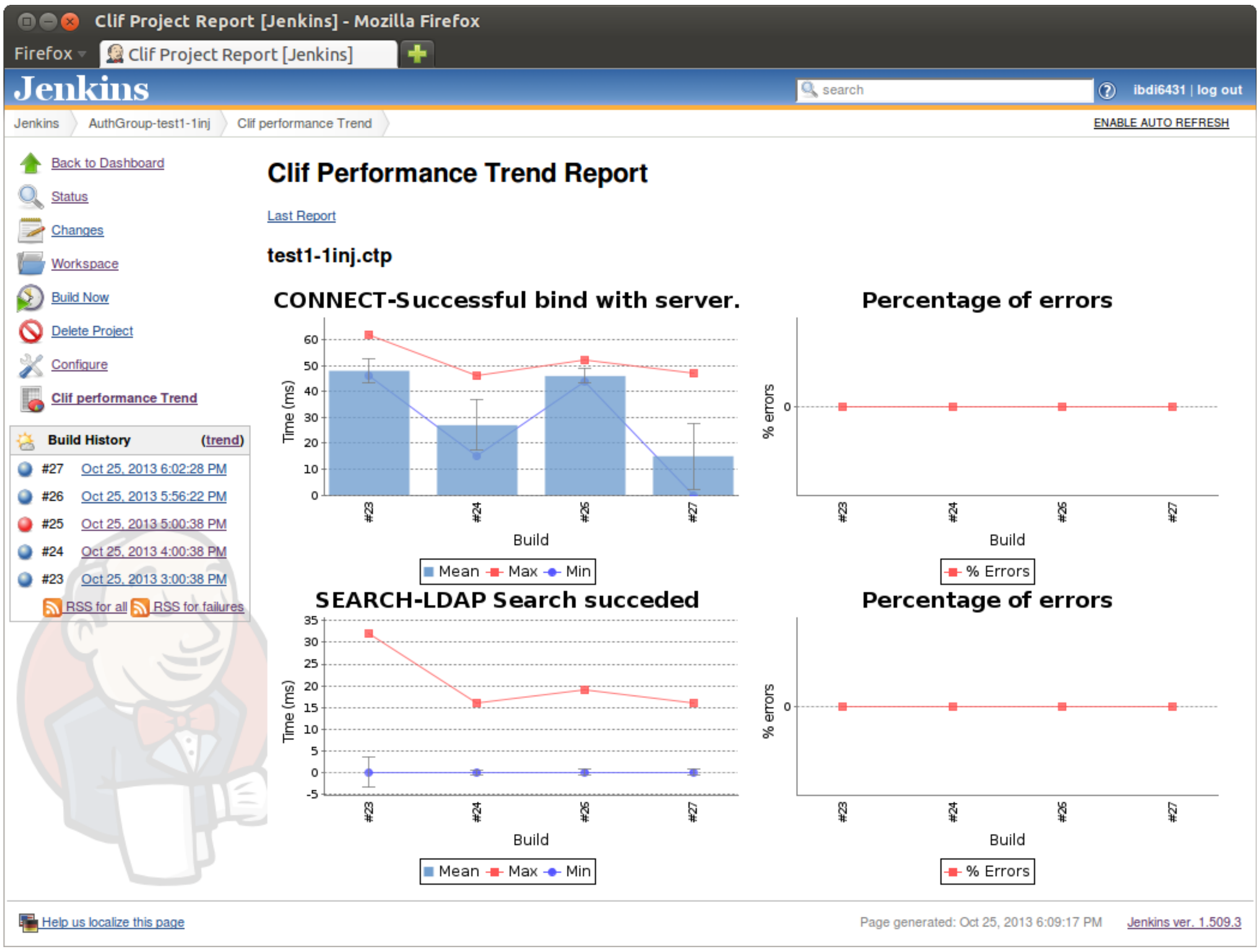
Injector - 0 - IsacRunner (Argument=random.xis Comment=)

Label	Samples	Avg	Median	Min	Max	Std	Throughput	% Errors
requete	8,872	168	160	32	334	60.28	148.42	0%

Aggregated values

Label	Samples	Avg	Median	Min	Max	Std	Throughput	% Errors
requete	17,745	126	116	30	334	63.37	296.86	0%

CLIF plug-in for Jenkins: Test-by-test detailed statistical analysis



CLIF plug-in for Jenkins: Performance trends through consecutive tests

9.2. How to create a CLIF test job

9.2.1. Rationale

Given a CLIF test project, consisting of a test plan, one or several scenarios and possible resource files (holding some data sets), we want to define a CLIF job in Jenkins that will gather those files in a Jenkins workspace and run the test.

There are two ways of achieving this result:

- going through a source code management system like CVS, SVN or GIT;
- going through a wizard that will take a CLIF project archive file (.zip).

9.2.2. Create a CLIF test job via SVN or GIT

As a continuous integration server, Jenkins is able to get all resources to run a build (or test) from a source code management system, like CVS, SVN or GIT.

First, the CLIF project must be first committed to a source code management system. If you are using the Eclipse-based console, we recommend to install and use an Eclipse plug-in for SVN or GIT.

Then, create the CLIF job in Jenkins : New Item > Freestyle project, choose a job/Item name, and click on button `OK`.

Then, in the job configuration page that follows, set the following:

- In the *Source Code Management* section, select your protocol and enter the repository information
- In the *Build* section, click on the Add build step and select either *Invoke Clif* or *Execute shell* or *Execute Windows batch command*. If you choose the *Invoke Clif* option, you will be using transparently the ant-based command-line interface, which is now discouraged. Otherwise, you will be using the new shell script-based command-line interface, which is recommended. The main interest is that it allows each CLIF job for having its own CLIF parameters (via file `clif.opts` in the CLIF test project), instead of sharing common parameters among all CLIF jobs using a given CLIF runtime (via file `etc/clif.props` in CLIF runtime).
  - if using the *Invoke Clif* option, you have to select the right CLIF runtime among those installed, and enter the test plan file name, relative to the Jenkins workspace directory, i.e. the root of the CLIF test project that will be fetched from the source code management system;
  - when using the shell script option, you just have to run the `clifcmd launch...` command for the right test plan, possibly followed by the `clifcmd quickstats` command if you are interesting in the *quickstats* report. Beware of adding the `bin` directory of a CLIF runtime to the path environment variable, or set the full path to the `clifcmd` command. Refer to sections 8.2.15 and 8.2.19 for details.
- In the *Post-build Actions* section, click on button *Add post-build action* and choose action *Publish Clif performance report*. A number of advanced options make it possible to customize CLIF reports.
- Click on button *Save* or *Apply*.

9.2.3. Import a CLIF test via the "wizard"

Once the CLIF plug-in for Jenkins is installed, the main menu features a new option: *Import a Clif zip*. Thanks to this wizard, you can directly create new CLIF test jobs from a CLIF test project archive file. The easiest way to build this archive file is to export the CLIF project from the Eclipse-based console. Otherwise, suffice to make a zip file holding the CLIF project directory, i.e. a directory containing all CLIF files: test plan(s), scenario(s), resource file(s), `clif.opts`, etc.

- Click on *Import a Clif zip*, choose the CLIF project archive file to import, and click on button *Import*[1].
- The wizard creates one job per test plan. Make your selection of test plans to import, choose the Clif runtime to use for these test plans, and click on button *Validate*.

Jobs are automatically created, including the build and post-build actions (respectively running the test and generating the CLIF report).

Note. The created jobs use the ant-based interface. However, it is still possible to edit the job and switch it to use the shell script-based interface (refer to section 9.2.2 above).

1 Careful with Jenkins' automatic refresh: when activated, the chosen file name may be discarded on refresh, and clicking on the *Import* button may raise an error. Should this occur, just try again or disable automatic refresh.

10. Test results and measurements

Whatever the user interface, CLIF tests gather the following data:

- test plan copy,
- Java system properties at test execution time for all probes and injectors,
- measurements from all probes and load injectors,
- life-cycle events for all probes and injectors,
- alarms generated by injectors or probes (if any).

All these data are gathered in a hierarchy of CSV[1]-files in a subdirectory of CLIF's runtime environment named "report" by default. This target directory may be changed with a system property (see Appendix A page 43).

Moreover, a so-called `quickstats` report may be automatically generated after each test execution. It gives a table of statistical results about response times, as well as throughput and number of errors for each type of issued request, globally for all load injectors. The `quickstats` feature is available:

- from the Eclipse console, by clicking on the collect button once a test is complete - the report is written to a file in a dedicated directory (`stats` by default) of the test's project;
- from the command line tools, using the `quickstats` command - the report is printed to the terminal's standard output (see section 8.2.19 or 8.3.19).

Both the Eclipse RCP-based console (section 6) and the Java Swing-based console (section 7) provide a graphical and statistical analysis tool. Moreover, the Eclipse-based console makes it possible to store monitoring moving statistics to files stored in the `stats` (by default) directory in the project's directory.

The CLIF plug-in for Jenkins automatically generates performance reports, test by test, and trends through tests (see section 9).

To sum-up:

user interface > available data	***command line tools***	***Swing console***	***Eclipse console***	***Jenkins***
raw measurements	yes	yes	yes	yes
`quickstats` reports	yes	no	yes	yes
moving statistics	no	no	yes	no
custom reports	no	yes	yes	no
automatic reports	no	no	no	yes

1 Comma-Separated Values, a common text-based export format for spreadsheet programs. Each line of the CSV file contains an event (measure entry), and values hold by each event are separated by a comma.

11. Licenses

CLIF is open source software licensed under the GNU Lesser General Public License (LGPL).

CLIF comes with facilities including the following open source software libraries:

- Jakarta commons Httpclient, from the Apache Software Foundation, released under Apache License;
- OpenLDAP from the OpenLDAP Foundation, released under OpenLDAP Public License
- Htmlparser from Source Forge, released under LGPL license;
- Eclipse graphical user interface libraries and Rich Client Platform, released under Common Public License;
- PostgreSQL JDBC driver, released under BSD license;
- DnsJava for DNS injection support, released under BSD License;
- JDOM for XML parsing in ISAC, released with a specific license.
- JavaMail for IMAP load injection, released under Berkeley license.

Appendix A: system properties

A number of Java system properties are set in file etc/clif.props of CLIF runtime environment. This file is used by the helper ant targets provided in file build.xml located at the root of CLIF runtime environment. Should you need to use CLIF without ant, don't forget to set all these system properties when launching the appropriate class in your Java Virtual Machine.

System properties used by CLIF are listed in the table hereafter:

system property	comment	default value in file etc/clif.props and clif.opts	default value in binary code
java.security.policy	set Java security policy file	etc/java.policy	*none*
fractal.provider	set Fractal implementation	org.objectweb.fractal.julia.Julia	*none*
fractal.registry.host	set hostname running FractalRMI registry. Most often, the registry in run by the the console (so the host is the console's host)	localhost	
fractal.registry.port	set port number for the FractalRMI registry launched by the console.	1234	
julia.config	using Julia as Fractal implementation, set Julia configuration file	etc/julia.cfg	*none*
julia.loader	using Julia as Fractal implementation, set class loader	org.objectweb. fractal.julia.loader .DynamicLoader	*none*
clif.codeserver.port	set port number for class and resource server embedded in the console	1357	*none*
clif.codeserver.host	set host name for class and resource server embedded in the console	localhost	
clif.codeserver.path	ordered set of directories where the codeserver may look for classes and resources it is asked for, separated by ; character. Note that, whatever the value of this property, classes and resources are first looked for in the jar files in lib/ext/ directory, and in the console's current directory. Absolute paths are used as is, while relative paths are interpreted from the root of CLIF's runtime environment.	examples/classes/ *(just to make examples run)*	*none*

system property	comment	default value in file etc/clif.props and clif.opts	default value in binary code
clif.codeserver.shared	When set to yes, true, on or 1 (case-insensitive), a single code server instance will be used for all further deployments, instead of one instance for each deployment. The shared code server is automatically launched when running the registry command.	*none*	false
clif.deploy.besteffort	When set to yes, true, on or 1 (case-insensitive), the deployment process discards from the test plan all blades that cannot be successfully deployed, for any reason (missing or mute CLIF server, blade failing to be deployed). Then, the resulting sub-test plan is regarded as successfully deployed as long as at least one injector or probe could be successfully deployed.	*none*	false
clif.deploy.retries	Number of retry attempts to reach a CLIF server for deploying new blades. This feature applies to CLIF servers that would be registered in the CLIF registry, but that would not be ready yet.	*none*	0
clif.deploy.retrydelay	Sleeping delay, in milliseconds, between two consecutive blade deployment attempts on a registered but unreachable CLIF server.	*none*	2000

system property	comment	default value in file etc/clif.props and clif.opts	default value in binary code
clif.deploy.timeout	Maximum acceptable time, in milliseconds, waiting for a set of blades to be deployed on each CLIF server. When the time-out is reached for one CLIF server, all of its blades, as defined in the test plan, are discarded. This resuts in a test plan deployment failure, or not, according the "best effort" setting. Note: this duration shall be consistent with, and actually greater than, the number of deployment attempts multiplied by the delay between two consecutive attempts (see the "retries" settings). 0 means no time-out (i.e. deployment time is unlimited).	*none*	0
clif.filestorage.clean	Sets the cleaning policy with regard to the conservation of test measurements. The "none"policy keeps all measurements of all tests. The "auto" policy discards all output from previous tests as soon as a new test is initialized.	"none"	"none"
clif.filestorage.delay_s	Sets the delay (in seconds) before the file-based storage system actually writes events after they are created. Typical value should be greater than the variation of response times to get events written in chronological order. However, this delay setting may be subsumed by the setting of maximum number of pending events (see property clif.filestorage.maxpending)	60	60

system property	comment	default value in file etc/clif.props and clif.opts	default value in binary code
clif.filestorage.maxpending	Sets the maximum number of pending events, waiting to be written to file because they are younger than the write delay (see property clif.filestorage.delay_s). Whenever this threshold is overwhelmed, oldest pending events are written without waiting for the write delay. This may lead to chronologically unordered events in the file, but prevents from saturating the JVM's heap memory because of event buffering (especially for high event throughputs).	1000	1000
clif.filestorage.dir	Sets the file system directory to be created (if necessary) and used to store the generated measures. An absolute path is used as is, while a relative path is interpreted from the root of CLIF's runtime environment.	*none*	report
clif.filestorage.host	sets a local IP address or a subnet number to be elected by the filestorage component when collecting events through TCP/IP sockets	*commented out, but set by "ant config"*	*random choice among locally available*

system property	comment	default value in file etc/clif.props and clif.opts	default value in binary code
clif.globaltime	Enables/disables global time reference on initialization. When enabled (property set to "yes", "true", "on" or "1", case-insensitive), events' dates are relative to the central time transmitted on initialization. In order to get an accurate date synchronization, a network time management protocol such as NTP should be active on all the CLIF servers' hosts. This property needs to be set only by the tool which performs the initialization call (i.e. not by the CLIF servers themselves). When disabled (property not set or set to any other value), events' dates are relative to local initialization time, which may differ from one blade to another, since initialization is triggered in a round-robin among all blades.	*false*	*false*
clif.isac.threads	Size of ISAC execution engine's pool of thread. The optimal value depends on the average requests throughput and the average response time.	100	100
clif.isac.groupperiod	update period (in ms) of active virtual users populations to match the specified load profiles	100	100
clif.isac.schedulerperiod	polling period (in ms) for the threads of the thread pool asking for something to do	100	100
clif.isac.jobdelay	When positive, gives the delay threshold (in ms) before an alarm is generated when a think time is longer than specified. -1 disables this feature.	-1	-1

system property	comment	default value in file etc/clif.props and clif.opts	default value in binary code
clif.quickstat.cleanfactor	floating point value. When strictly greater than zero, enables measures cleaning when generating the "quickstats" statistical report: response times greater than (mean + f*standard deviation) or less than (mean - f*standard deviation) are discarded. When less than or equal to zero, all measures are kept.		*0*
clif.quickstat.cleanlimit	floating point value between 0 and 100. When measures cleaning is enabled for the "quickstats" report, this setting gives the minimum percentage of measures to keep.		*95.4*
jonathan.connectionfactory .host	sets a local IP address or a subnet number to be used by the FractalRMI remote object references	*commented out, but set by "ant config"*	*random choice among locally available*

Other system properties may be useful for a variety of use cases (they are given in comments in file `etc/clif.props.template`):

- for remote Java debugging:
  `-agentlib:jdwp=transport=dt_socket,address=8000,server=y,suspend=n`
- for SSL certificates (for example for HTTPS support):
  ```
  -Djavax.net.ssl.trustStore=/path/to/keystore
  -Djavax.net.ssl.trustStorePassword=the_keystore_password
  ```

Appendix B: Class and resource files (remote) loading

Principle

When components are deployed in a CLIF server (probe, injector), the corresponding classes are automatically downloaded from the console if they are locally missing. Moreover, those components may require resource files (see `webtest.urls` file in `webtest` example, or `helloworld.xis` file in `isac-helloworld` example), which the user would rather not have to copy on every CLIF server. The content of these resource files can be remotely read via the console too.

This feature relies on a specific Java class loader and its associated system property `clif.codeserver.path` on the one hand, and on a so-called "code server" embedded in the console on the other hand.

Where classes and resource files are looked for?

The code server embedded in the console looks for the requested classes and resources successively in the following places:

- jar files in CLIF distribution's lib/ext/ directory where the console is running. Note: since the code server indexes the contents of all jar files in lib/ext/ at console start-up, all necessary jar files must be present before running the console;
- the console's current directory (which should be CLIF's root directory);
- the directories declared by `clif.codeserver.path` property, relative to the console's current directory.

See appendix on system properties page on User Manual for details on how to set the `clif.codeserver.path` property, and how to set the port number for the code server.

Appendix C: ISAC scenario DTD

```
<!-- A scenario is composed of two parts :-->
<!--  - behaviors, to define some behavior...-->
<!--  - load, to define the load repartition...-->
<!ELEMENT scenario (behaviors,loadprofile)>
<!-- In the part behaviors, we must define the plugins that will be used in
behaviors-->
<!ELEMENT behaviors (plugins,behavior+)>
<!-- For each plugin we define the plugin with the use tag-->
<!ELEMENT plugins (use*)>
<!-- We can add some parameters if it's needed-->
<!ELEMENT use (params?)>
<!-- We define an id which can be used in the next parts, to reference the
plugin used-->
<!-- The name is the name of the plugin that will be used-->
<!ATTLIST use
  id        ID     #REQUIRED
  name      CDATA  #REQUIRED
>
<!-- Now we can define the behaviors-->
<!-- a behavior begin with the behavior tag, and can be composed of: -->
<!--   - A sample : reference to a specified sample plugin... -->
<!--   - A timer : it's a reference to a timer plugin... -->
<!--   - A while controller : it's a while loop... -->
<!--   - A preemptive : it's a controller adding a preemptive for all it
children... -->
<!--   - An if controller : it's a controller doing the if / then /else task...
-->
<!--   - A nchoice controller : it's a controller which permits doing random
choices between some sub-behaviors with a weight factor -->
<!ELEMENT behavior (sample|timer|control|while|preemptive|if|nchoice)*>
<!-- When we define a behavior we must define the id parameter too, -->
<!-- it will be used to reference behavior in load part-->
<!ATTLIST behavior
  id        ID     #REQUIRED
>
<!-- A sample element could need some parameters-->
<!-- the parameters needed are defined in the plugin, which will be used,
definition file-->
<!ELEMENT sample (params?)>
<!-- A sample element have for parameter : -->
<!--  - use : the id of the plugin that will be used for this sample-->
<!--          the id of this plugin must be defined into the plugins part-->
<!--  - name : the name of the action that is referenced by the sample tag-->
<!--           this action name must be specified in the plugin, which is used,
definition file-->
<!ATTLIST sample
  use       CDATA  #REQUIRED
  name      CDATA  #REQUIRED
>
<!-- A timer element could need some parameters-->
<!-- the parameters needed are defined in the plugin, which will be used,
definition file-->
<!ELEMENT timer (params?)>
<!-- The timer have got the same parameters of a sample element-->
<!ATTLIST timer
```

```
  use       CDATA  #REQUIRED
  name      CDATA  #REQUIRED
>
<!ELEMENT control (params?)>
<!ATTLIST control
  use       CDATA  #REQUIRED
  name      CDATA  #REQUIRED
>
<!-- A while controller must contain a condition and a sub-behavior-->
<!ELEMENT while (condition,(sample|timer|control|while|preemptive|if|nchoice)*)>
<!-- A condition is a reference to a test of a specified plugin-->
<!-- it could need some parameters-->
<!ELEMENT condition (params?)>
<!-- we need specified as parameters for this tag, the plugin used and the name
of the test, like sample or timer tag-->
<!ATTLIST condition
  use       CDATA  #REQUIRED
  name      CDATA  #REQUIRED
>
<!-- A preemptive element is defined as a while element, the difference is in
the execution process-->
<!-- For a while we evaluate the condition before each loop, in a preemptive
before each action...-->
<!ELEMENT preemptive (condition,(sample|timer|control|while|preemptive|if|
nchoice)*)>
<!-- An if controller must contains a condition and a sub-behavior ('then'
tag)-->
<!-- And optionally it could contain another sub-behavior ('else' tag)-->
<!ELEMENT if (condition,then,else?)>
<!-- A then tag delimited the sub-behavior that will be executed if the
condition is true-->
<!ELEMENT then (sample|timer|control|while|preemptive|if|nchoice)*>
<!-- A else element contains a sub-behavior too-->
<!ELEMENT else (sample|timer|control|while|preemptive|if|nchoice)*>
<!-- A nchoice plugin contains n sub-behavior, each sub-behavior have a
probability to be executed-->
<!ELEMENT nchoice (choice+)>
<!-- An choice element contain a sub-behavior-->
<!ELEMENT choice (sample|timer|control|while|preemptive|if|nchoice)*>
<!-- And this element take for parameter a probability-->
<!ATTLIST choice
  proba    CDATA  #REQUIRED
>
<!-- Now we define the params element, this element begin the part to define
parameters for the parent element-->
<!ELEMENT params (param+)>
<!-- For each param we need to define it with the param tag-->
<!ELEMENT param EMPTY>
<!-- This tag take for parameters the name of the parameter and it value-->
<!ATTLIST param
  name      CDATA  #REQUIRED
  value     CDATA  #REQUIRED
>
<!-- Now let's define the load part, this part is used to define the ramps, each
ramps represent the load for a behavior-->
<!-- We can define some ramps together in a group element, this element is used
to launch several behaviors in the same time-->
```

```
<!ELEMENT loadprofile (group*)>
<!-- A group is a composition of 'ramp' elements-->
<!ELEMENT group (ramp+)>
<!-- We need define the behavior id of the group and optionally -->
<!-- the force stop mode, default is true -->
<!ATTLIST group
  behavior        CDATA        #REQUIRED
  forceStop         (true|false) "true"
>
<!-- each ramp could take some parameters-->
<!ELEMENT ramp (points)>
<!-- For a ramp we must define the style of the ramp, which will be used-->
<!ATTLIST ramp
  style     CDATA  #REQUIRED
>
<!ELEMENT points (point,point)>
<!ELEMENT point EMPTY>
<!-- For a ramp we must define the style of the ramp and the reference of the
behavior, which will be used-->
<!ATTLIST point
  x     CDATA  #REQUIRED
  y     CDATA  #REQUIRED
>
```

Appendix D: ISAC execution engine

The ISAC execution engine is the interpreter class for ISAC scenarios. When editing a test plan, just select the “injector” role and type `IsacRunner` in the “class” field. Then, fill the “arguments” field with the file name of the ISAC scenario you want to run. As a general advice, don't set the full path name but simply the file name, and add the directory where the scenario file resides to the code server path (see appendix p. 49). When using the Eclipse console, the file typically resides in the project directory.

The ISAC thread pool

The ISAC execution engine uses a pool of threads to run virtual users (aka behavior instances). When a virtual user is engaged in a think time, its execution thread is used to activate another virtual user. This way, the size of the thread pool is typically far smaller than the maximum of simultaneously running virtual users that is specified by the load profile. This pool has a default size that may be changed:

- before runtime:
  - either by setting system property `clif.isac.threads`
  - or by adding option `threads=my_custom_pool_size` in the “arguments” field;
- at runtime, by changing the value of parameter “threads”.

Millions of virtual users per execution engine can easily be reached. The issue is that the think times must be much greater than the response times in order to really support such a number of virtual users without violating the specified behaviors. The theoretical optimal thread pool size is:

$$\text{optimal pool size} = \frac{\text{maximum number of virtual users} * \text{average response time}}{(\text{average think time} + \text{average response time})}$$

The actual optimal pool size shall be a little greater to face possible transient variations of the global activity (when many virtual users simultaneously exit from a think time) and the overhead of context switching between virtual users. The default size (see appendix p. 43) may require to be adjusted to your particular test case. Of course, setting an over-sized pool of threads will waste computing resources and result in performance degradation.

Deadline violation alarms (Job delay)

When the execution engine becomes overloaded, a consequence is that virtual users' think times become longer than specified. In other words, the deadline for performing the action next to the think time is violated. It is possible to get an alarm event when a given tolerance threshold is reached. This feature is enabled as soon as a positive value is set for this threshold, expressed in milliseconds. To set the threshold:

- before runtime:
  - either set system property `clif.isac.jobdelay`
  - or add option `jobdelay=my_custom_threshold_in_ms` to the “arguments” field;
- at runtime, by changing the value of parameter “jobdelay”.

Note that enabling this alarm results in a slight overhead in the execution engine functioning. Moreover, setting a small threshold value may result in a profusion of meaningless alarms: a small deadline violation from time to time does not necessarily mean the engine is overloaded. The relevant threshold value depends a lot on your use case, but a 100ms to 1000ms delay is probably a good order of magnitude. However, when analyzing the meaning of such an alarm, be careful also about the Java garbage collector that blocks the JVM and may cause deadline violations.

The default value is -1 (disabled).

Group period

The execution engine periodically checks if the current number of virtual users matches the specified load profile: in case some virtual users are missing, new ones are instantiated; in case virtual users are too numerous, some of them are stopped once their current action is complete. Stopping virtual users before the normal completion of their behaviors is performed only if the "force stop" option has been enabled in the load profile definition. Otherwise, the execution engine will just wait for the population to naturally decrease as behaviors complete.

The population checking period is set in milliseconds:

- before runtime:
  - by setting system property `clif.isac.groupperiod`
  - or by adding option `groupperiod=my_custom_group_period_ms` to the "arguments" field;
- at runtime, by changing the value of parameter "groupperiod".

The good period value is a trade-off between performance and accuracy of the engine: a short period will increase the engine overhead but the virtual users' population will be closer to the load profile specification. The default period (see appendix p. 43) is probably a good order of magnitude for common test cases.

Scheduler period

When a thread from the pool has just completed an action for a virtual user which is entering a think time period, it asks the engine for an action to do for another virtual user. If there is nothing to do at this time, the thread makes a small sleep before asking again, and so on until it gets something to do. The small sleep duration is given in milliseconds by the scheduler period parameter. This parameter may be changed:

- before runtime:
  - by setting system property `clif.isac.schedulerperiod`
  - or by adding option `groupperiod=my_custom_scheduler_period_ms` to the "arguments" field;
- at runtime, by changing the value of parameter "schedulerperiod".

The good period value is a trade-off between engine reactiveness and performance. A zero value should be avoided since the threads waiting for something to do would enter a frenetic polling loop on interrogating the engine, which typically wastes all processing power. A big value should be avoided too for the sake of think times accuracy. The formula below gives the possible variation range of think times:

$$\text{specified think time} \leqslant \text{actual think time} \leqslant \text{specified think time} + \text{scheduler period} + \text{context switching overhead}$$

The default value (see appendix p. 43) seems to be a good value for common test cases. In the general case, you should ensure that: (1) the scheduler period is significantly less than the think times, and (2) the scheduler period is significantly less than the job delay setting (when positive/enabled).

Storage options

As a CLIF load injector, the ISAC execution engine produces a number of events:

- one life-cycle event is produced each time the engine state changes: initializing, initialized, starting, running, suspended, etc. (refer to the CLIF Programmer's Guide for details about the blade life-cycle specification);
- one action event is produced for each request (aka sample) on the SUT;
- one alarm event may be generated each time a think time is actually longer than specified, according to the given tolerance threshold (see Job delay parameter described above).

These events are stored unless you specify not to do so, through the following parameters:

- `store-lifecycle-events`
- `store-action-events`
- `store-alarm-events`

Acceptable enabling values are: on yes true 1

Acceptable disabling values are: off no false 0

For action events, filter values are also acceptable formatted as bellow:

```
<keep>field:value
```

- Multi-value filter can be set, separated by pipe character | .
- `<keep>` values are:
  - + to specify action events to keep
  - - to specify action events to discard
- Acceptable field values correspond to fields in action event: `type success duration comment result`
- Value is a regex (see java.util.regex.Pattern)

Examples:

+success:false|-comment:optional	will keep failed action where comment is not equal to optional
-type:http\sget	will keep all except "http get" action

Disabling storage for an event type has the following advantages: increased ISAC engine power, reduced time for final data collection, reduced storage space. As a matter of fact, some test cases may generate gigabytes of data that may be too heavy to analyze. Moreover, high events throughputs (thousands of events per second) may overwhelm the disk transfer rate. The drawback of disabling event storage is that you won't keep any data for this event type on this injector.

A possible smart use of this feature is to disable action events storage for some massive load injectors (heavy background load), but to store and analyze the results from a couple of load injectors generating a light load. This way, you get a reduced amount of data, and data is quite accurate because the corresponding load injectors were far from saturating.

Note that disabling storage of life-cycle events and alarm events is possible but not recommended in common test cases:

- life-cycle events give an interesting and very lightweight trace of the injector's activity steps, whatever the test duration, with no noticeable impact on the engine performance;
- the occurrence of alarm events shows that something did wrong during the test, which is key to the test analysis, while no alarm event is generated when everything goes well.

As a conclusion, storage of life-cycle and alarm events is commonly always useful and never disturbing.

Dynamic load profile change

In case your scenario defines no load profile, or when you want to dynamically change the predefined load profile while a test is running, you can change parameter "population" of the ISAC execution engine. This parameter has the following form: $b_1=n_1;b_2=n_2;...$ where b_1 is the name of a behavior in the ISAC scenario and n_1 is the number of instances (aka virtual users) of this behavior.

When getting the current value of "population" parameter, if the current population is ruled by a specified load profile, you will get empty values: $b_1=;b_2=;...$ Since the population may change accordingly to the load profile, no value is given. Once a population is set for a behavior, the population for this behavior becomes constant and the load profile for this behavior is definitively lost. As a result, the test will never complete by itself: you will have to stop it by yourself, at the moment that seems relevant for you.

Note that increasing a behavior's population through the setting of "population" parameter should be made carefully: all necessary new virtual users are created at once, and may result in a brutal load increase on your injector and SUT. Depending on the desired effect, it might be wise to add a linearly distributed random think time at the beginning of your behavior definition so that virtual users don't simultaneously start their actual load activity even though their are created at the same time. Of course, you must anticipate on this when writing the scenario.