

ULCLoad 3.1

User Guide

Canoo Engineering AG
Kirschgartenstrasse 5
CH-4051 Basel
Switzerland
Tel: +41 61 228 9444
Fax: +41 61 228 9449
ulc-info@canoo.com
<http://ulc.canoo.com>

Copyright 2000-2012 Canoo Engineering AG
All Rights Reserved.

DISCLAIMER OF WARRANTIES

This document is provided "as is". This document could include technical inaccuracies or typographical errors. This document is provided for informational purposes only, and the information herein is subject to change without notice. Please report any errors herein to Canoo Engineering AG. Canoo Engineering AG does not provide any warranties covering and specifically disclaim any liability in connection with this document.

TRADEMARKS

Java and all Java-based trademarks are registered trademarks of Sun Microsystems, Inc.

All other trademarks referenced herein are trademarks or registered trademarks of their respective holders.

Contents

Overview	4
Introduction	4
Requirements	4
Resources	4

Installation	5
Getting a License	5
Downloading and Installing the Release	5
Release Structure	6

Load Testing How Tos	7
How To Start the ULC Load UI	7
How To Record a Test Plan	7
How To Rename Samplers	10
How To Repeatedly Record a Specific Use Case	11
How To Review a Test Plan	12
How To Play a Test Plan	14
How To Review Test Results	17
How To Analyze Round-Trips	19
How To Parameterize a Test Plan	22
How To Handle Extensions	23
How To Handle Custom Launchers	24
How To Run a Distributed Load Test	26
How To Use Client Certificates	27

Best Practices	29
Add Metadata to your ULC Application	29
Record Robust Scenarios	29
Play what you Record	29
Failed Samplers are Serious	29
Ramp Up Slow	30
Let the Virtual Users Think	30

Architecture	31
Recording	31
Replaying	32

Overview

Introduction

ULC Load is a tool to create load on ULC based applications. During a test run, ULC Load logs various performance data. The logged performance data lets you judge the scalability of your ULC application.

In particular, ULC Load enables you to:

- Record a scenario consisting of a sequence of user interactions.
- Review the recorded scenario with a graphical user interface.
- Parameterize the recorded scenario, e.g. parameterize the user name and password of a login dialog.
- Replay the parameterized scenario with an arbitrary number of virtual users.
- Measure and analyze performance data.

ULC Load integrates into Apache Jakarta's JMeter load test infrastructure. Therefore, it leverages features of JMeter, such as data visualization and analysis of load test results, and simplified load test distribution on multiple load driver machines to increase the load.

Requirements

- **Java Runtime**
ULC Load needs at least JRE 1.5.0
- **ULC**
ULC Load needs at least Canoo RIA Suite Update 5. ULC Load does not run with earlier ULC versions.
- **Servlet Container**
The ULC application load tested by ULC Load needs to be deployed as a Servlet container. A scenario where the ULC application is deployed as a stateful session bean in an EJB container is currently not supported.

Resources

- ULC
<http://ulc.canoo.com>
- Jakarta JMeter
<http://jakarta.apache.org/jmeter>

Installation

This section explains how to install ULC Load. The tasks involved are:

- Getting an evaluation or developer license for ULC Load
- Downloading and installing the release

Getting a License

Get an evaluation or developer license for ULC Load from: <http://www.canoo.com/kurt>.

Make sure that you have downloaded and installed the right version of ULC as specified in the section **Requirements**.

Downloading and Installing the Release

To install the release:

- Register with Canoo Customer Portal <http://www.canoo.com/kurt> to be able to order a license key and download the installer.
- Download and run the installer corresponding to your platform. The **Introduction** dialog box displays.
- Click **Next**.
- Select **I accept the terms of the License Agreement** and click **Next**.
- The **License Information** dialog box displays. Copy the license key you received by email from ---START LICENSE LIST--- to ---END LICENSE LIST--- and paste this key into the **License Information** text box.
Please note: in some cases your email client may introduce line breaks changing the format of the license key email. Please make sure that the each line of the license key text follows the format *name=value*, like in a Java Properties file.
- Click **Next**.
- Select the folder in which you would like to install ULC Load and click **Next** to continue.
- Select the folder where you would like to place the shortcuts and click **Next**.
- Specify the folder where you have installed ULC. For example, on Windows: C:\Program Files\canoo-ria-suite. ULC Load uses client side libraries from your ULC installation to ensure version compatibility with ULC libraries used by your application on the server.
- Review the **Pre-Installation Summary**. Click **Next** to start the installation process.
- Click **Done** to quit the installer.

Release Structure

The current release of ULC Load has the directory and file structure outlined below.

ulcload-3.1	ULC Load directory.
bin	ULC Load binaries.
base	ULC Load's core library including dependencies and sources required for supporting custom application recorders. (See section How To Handle Custom Launchers for instructions on how to create a custom recorder for your application.)
doc	ULC Load documentation.
ui	ULC Load's graphical user interface for scenario recording and replay.
previous_releasenotes	Previous release notes for reference purposes.

Load Testing How Tos

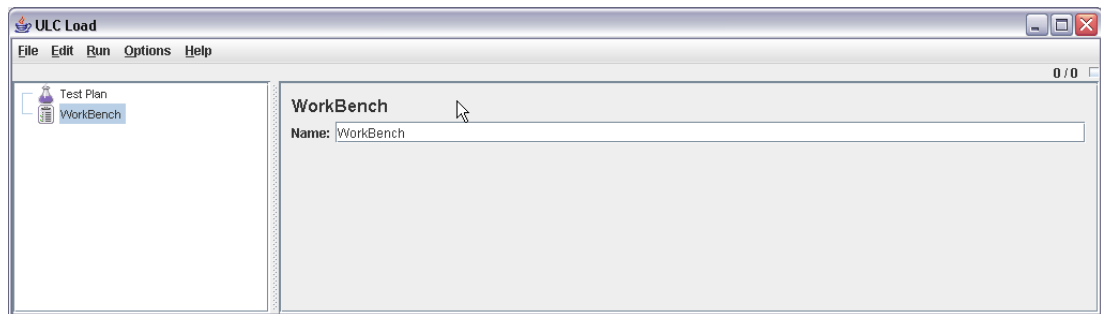
This section describes how to use the ULC Load user interface to run load tests for your ULC application. Both the ULC Load user interface and the load-testing infrastructure are based on Apache JMeter. Apache JMeter and the necessary extensions to load test ULC applications inside Apache JMeter are installed as part of the ULC Load release.

Please refer to <http://jakarta.apache.org/jmeter/usermanual/index.html> for detailed information about Apache JMeter.

How To Start the ULC Load UI

- Open the **Start** or **Program** menu and select **ULC Load 3.1 > ULC Load UI** to start the graphical user interface of ULC Load.

This opens the following window:

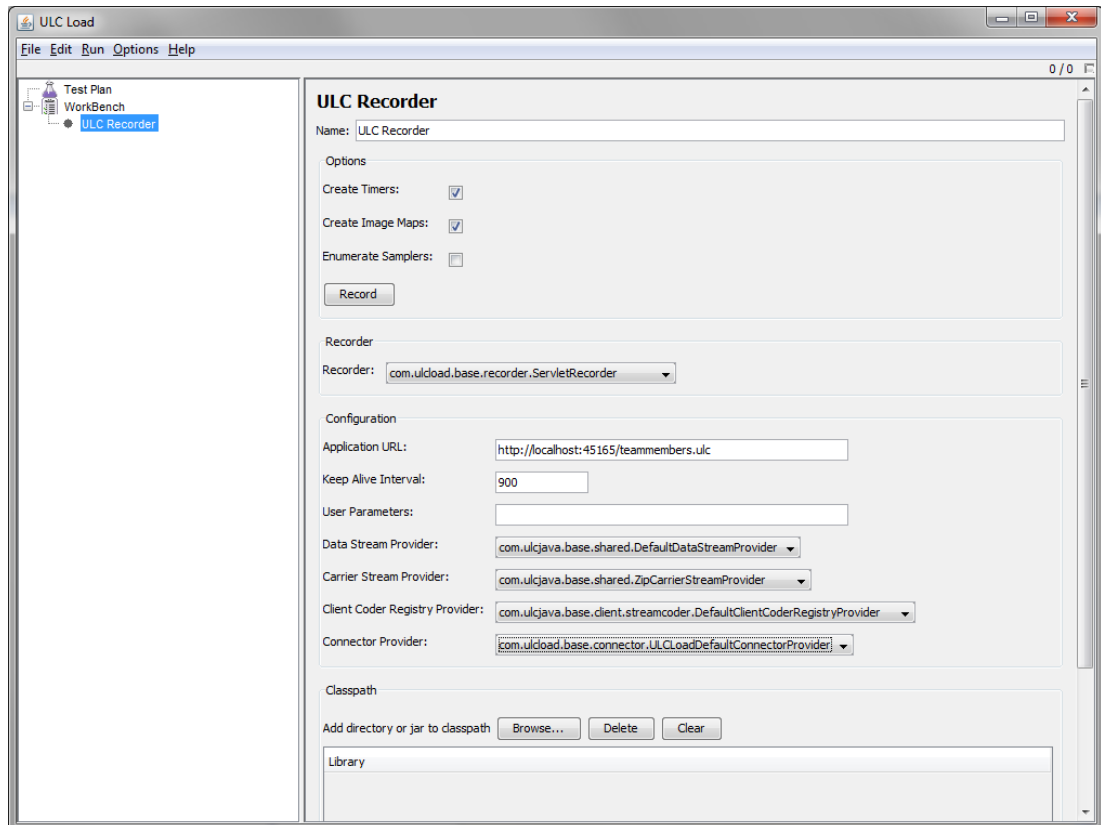


The **Test Plan** node comprises steps executed by ULC Load when it is run. A complete test plan will consist of one or more Thread Groups, Logic Controllers, Sample Generating controllers, Listeners, Timers, and Configuration Elements. These elements can be added or removed in a Test Plan using the right-click menu. The **WorkBench** node is a container for temporary work items. When saving a test plan only the nodes below the **Test Plan** node are saved, all nodes below the **WorkBench** node will be discarded when you close ULC Load.

How To Record a Test Plan

This section describes how to record a test plan for a ULC application. For the purpose of demonstration we will work with the TeamMembers sample application that is part of the ULC release. To start the TeamMembers application in Tomcat, open the **Start** or **Program** menu and select **Canoo RIA Suite > Sample Applications > Start Sample Application Server**.

- Use the context menu to add a new **Non-Test Elements > ULC Recorder** to the existing **Workbench** tree node.



The **ULC Recorder** is used to record the round trips (within a usage scenario) of an ULC application.

In the recorder, you can specify whether or not you want to create **Timers** between the requests. **Timers** will cause ULC Load to introduce time delays between the requests during replay. This time delay is equal to the time delay between the requests at the time of recording.

You can also specify if the recorder should record image maps. Image maps help you to analyze the recorded test plan. You can read more about image maps in section **How To Review a Test Plan** below.

The third option allows you to enumerate samplers, which makes unique names for the samplers created during recording. Should you forget to check that option and want to make the sampler names unique after recording, you can still do so using the **Sampler Relabeler** (see later in this chapter).

ULC Load creates a test plan skeleton when recording a scenario. The test plan skeleton contains all required test elements such as a **Thread Group**, a **ULC Replay Controller**, a **ULC Session Manager** and various reporting elements.

Various parameters for the ULC client can be specified on the recorder. They are:

- **Recorder class**
Default is **com.ulcload.base.recorder.ServletRecorder**.
You can read more about Recorder classes in section **How To Use Custom Launchers** below.

- **Application URL**
The URL of the ULC Application Servlet.
In this example we will use the URL of the TeamMembers sample that we started earlier: <http://localhost:45165/teammembers.ulc>.
- **User Parameters**
Example: firstName=firstValue, secondName=secondValue.
- **Data stream provider class**
- **Carrier stream provider class**
- **Client coder registry provider class**
- **Connector provider class**

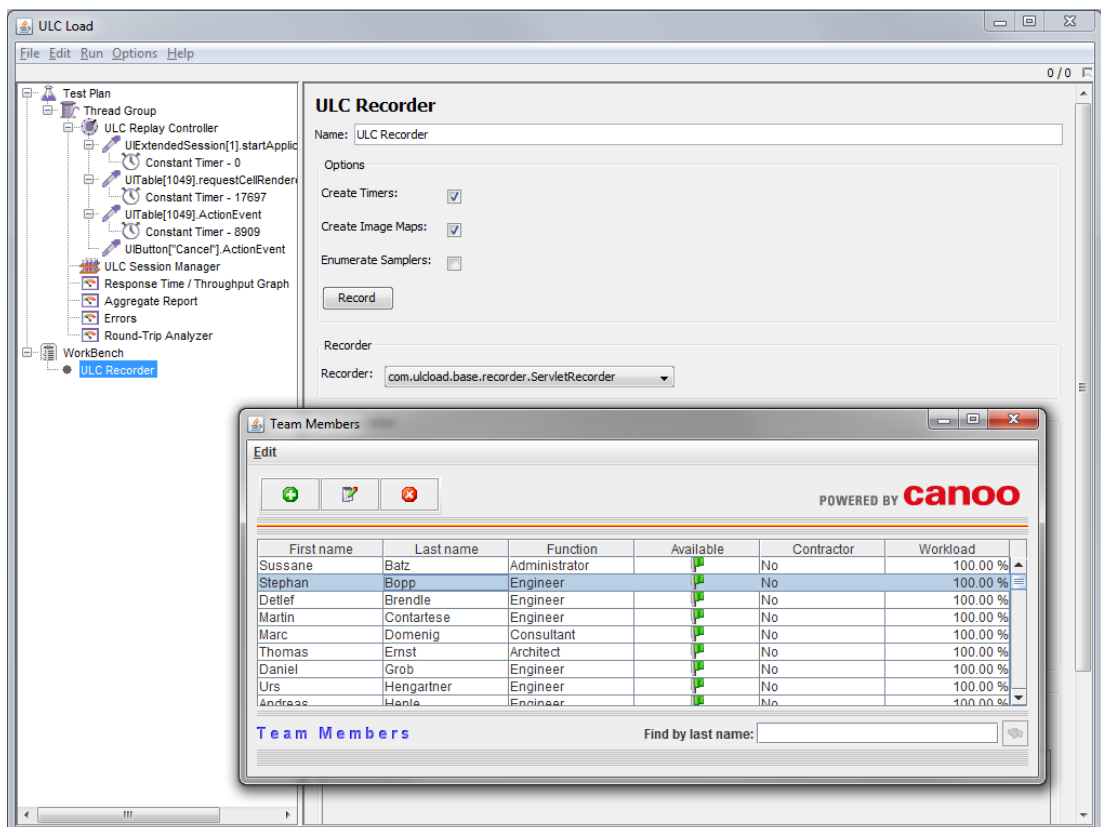
The Connector provider class is an implementation of IConnectorProvider and it provides a connector that is used by ULC Load recorder and player to connect to the ULC Application.

com.ulcload.base.connector.ULCLoadConnectorProvider is the default implementation which provides ULC's ServletConnector.

Please see ULC documentation for details on the above parameters to the ULC client.

If the combo boxes are left blank, corresponding default parameters in ULC are applied.

- Press the **Record** button in the **ULC Recorder** element to record a test plan. The TeamMembers application starts and you can start interacting with it to create requests that will be captured by the recorder.



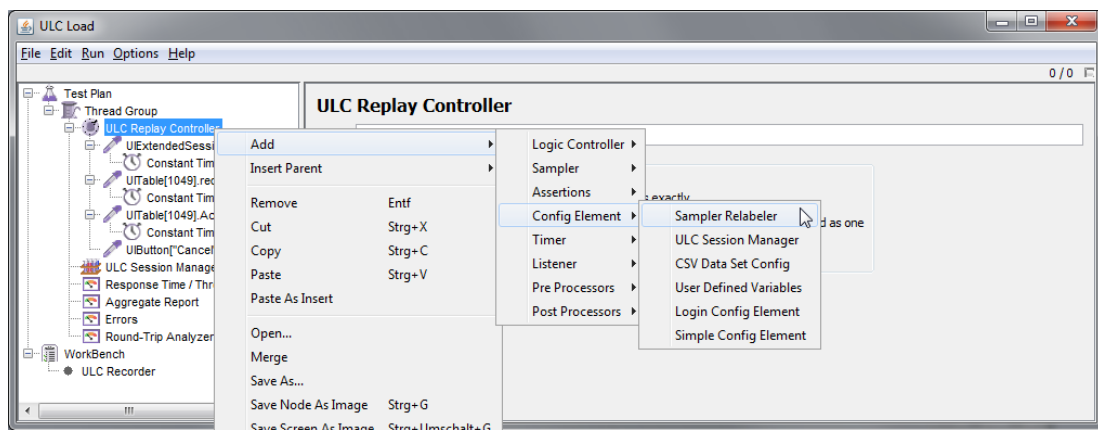
Once you are finished with your user interaction scenario, you can terminate the TeamMembers application by closing the application window.

The requests generated during the scenario have now been recorded in the **ULC Replay Controller**. The **ULC Replay Controller** acts as a container for recorded **ULC Samplers**.

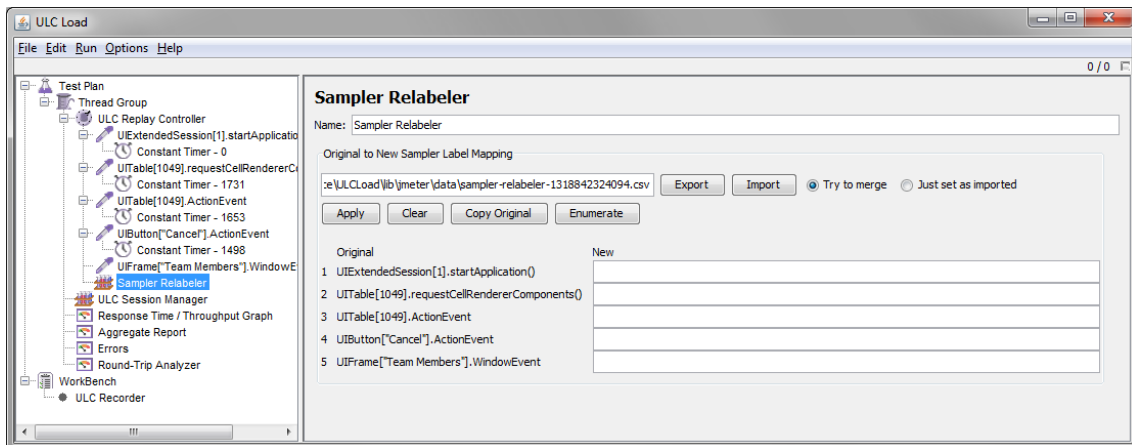
How To Rename Samplers

Sometimes the default sampler names given by ULCLoad are not what is needed. One reason could be that after load testing the **Aggregate Report** shall hold a separate row for every sampler, even though multiple samplers have equal names. Another reason could be that samplers need names that are less generic and thus easier to understand for anyone who is familiar with the application under test, but not with ULC. Whatever the reason may be, samplers can be renamed one by one by selecting them and then entering a new name. However this is quite cumbersome if many or all samplers need to be equipped with new names.

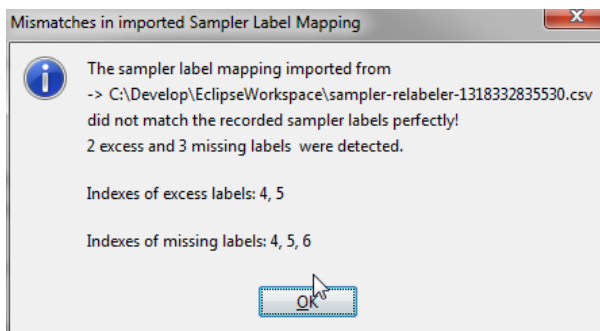
A more comfortable approach is to add a **Sampler Relabeler** (under Config Element) to the **ULC Replay Controller**.



The Relabeler GUI allows you to enter new sampler names and quickly switch from one entry field to the next using the tab key. The new names can be based on the current (original) ones by pressing the **Copy Original** button. In order to enumerate the names, press the **Enumerate** button. When satisfied with the new names, apply them to the samplers pressing the **Apply** button. Using these three buttons in a row makes the original sampler names unique exactly as if you would have checked the **Enumerate Samplers** option on **ULC Recorder** prior to recording.



You may also export the sampler label mapping (original to new) in CSV format either for later (re-)use or in order to modify it using other tools (for example Windows Excel). Of course such mapping files can then be imported as well. Two import modes are available. The simpler mode just reads the new sampler names and applies them to the entry fields (not yet to the samplers!) in the order of reading. If the current test plan is very close to but not exactly the same as the test plan on which the label mapping is based upon, you will get a better result when choosing the merge option. This option tries to identify matching original sampler names and applies the new names accordingly. If the current test plan has additional samplers, the corresponding entry fields are kept empty. If the current test plan has fewer samplers, the corresponding new names from the mapping are ignored. If the original sampler names didn't match perfectly, a short report is displayed informing you about which imported names didn't match any of the current sampler names (excess labels) and for which of the current sampler names no imported name was found (missing labels).

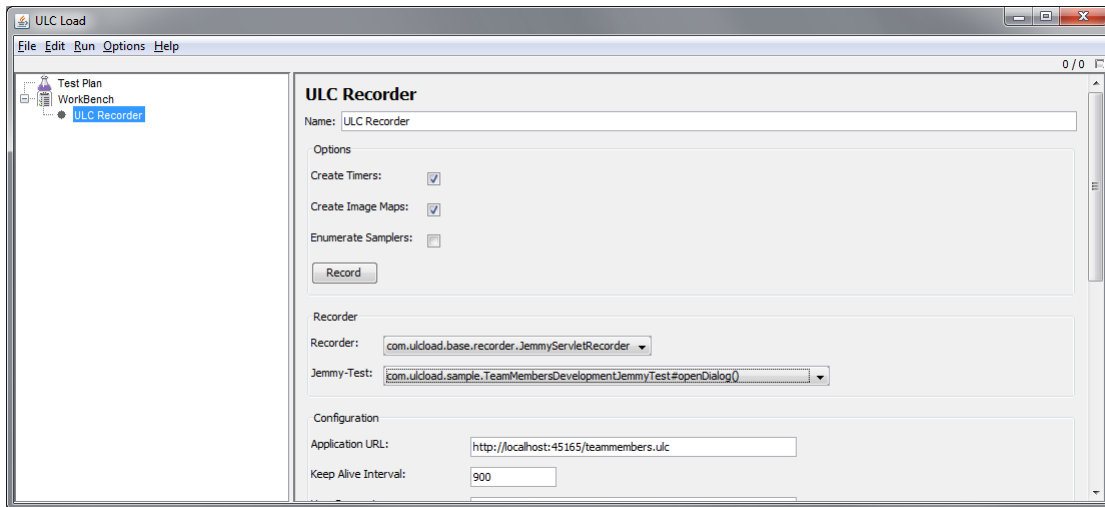


How To Repeatedly Record a Specific Use Case

Should you need to record a specific use case again and again, you can use Jemmy tests to drive the recording. A typical scenario in which this approach makes sense is the following: an application is developed iteratively and on each such iteration the application shall be load tested, however the application changes are significant enough to invalidate the test plan from the previous iterations.

As a prerequisite you need a Jemmy test that is a subclass of *com.ulcjava.testframework.AbstractTestCase* (see the ULC Test Framework Guide) on ULCLoad's classpath. Add the corresponding directory or jar file if necessary. Then,

recording is just three clicks in **ULCRecorder** away: first choose the **JemmyServletRecorder**, then the **Jemmy-Test**, and finally press the **Record** button.



An additional note regarding timers: as Jemmy tests typically don't incorporate user think time, the JemmyServletRecorder adds a single **Random Timer** to the **ULC Replay Controller** instead of adding **Constant Timers** to the samplers (see **Let the Virtual User Think** for reasons why it's advisable to incorporate user think time). This way you have the better of both worlds: reliable and fast recording on one hand, realistic load while testing on the other.

How To Review a Test Plan

Follow these steps to review a test plan.

- Take a look at the **ULC Replay Controller** node. Each item under this node is a **ULC Sampler** or a **Timer** clock.
- Select a **ULC Sampler** node and review the server roundtrip. A **ULC Sampler** sends request to the server which results in one or more server roundtrips. These server roundtrips appear as list of expressions.
- The upper section of the detail view shows the expressions that are part of the **ULC Sampler**. Expressions are of 4 types:

➡ Represents a method invocation from server to client

➡ Represents an event from client to server

⊕ Represents a new proxy request from server to client

⬅ Represents a method invocation from client to server

You can find more information about expressions in the chapter "Architecture".

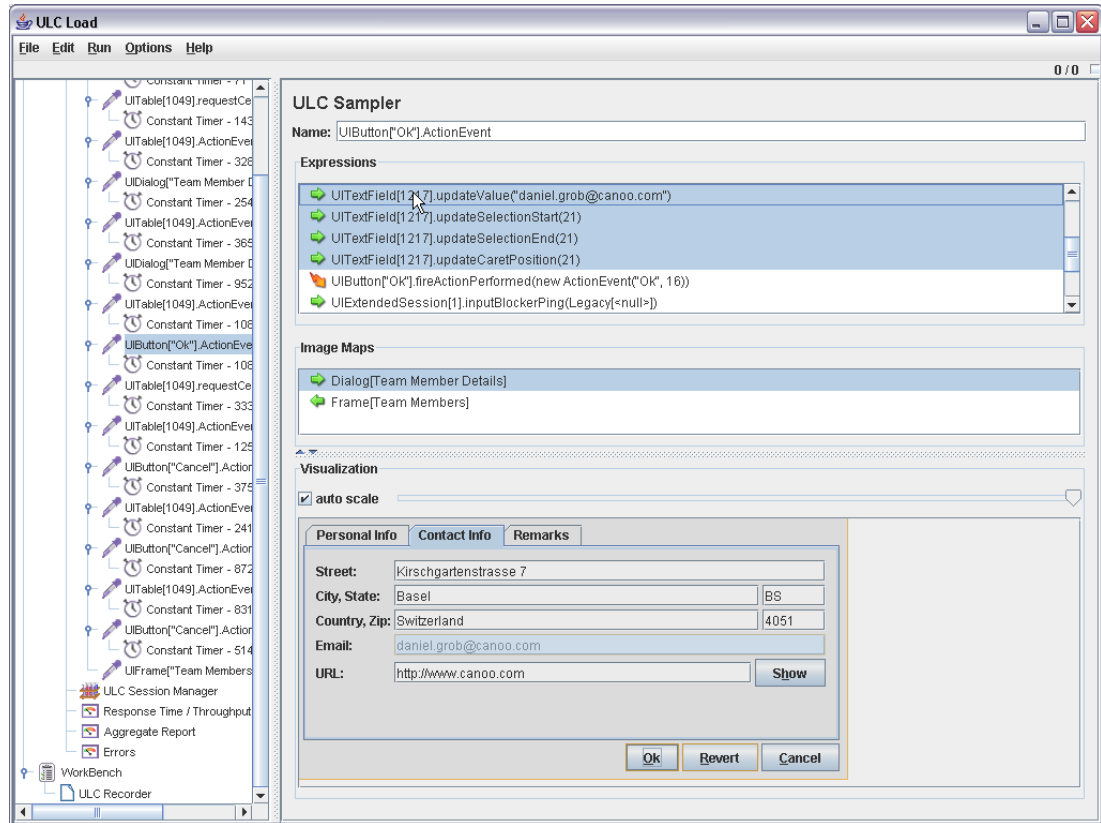
- Select an **Expression** and look at the expression details.

Each sampler has image maps associated with (only in case you decided to capture image maps as part of the recording). Image maps are of 2 types:

➡ Represents an image map that was created **before** the round trip started

⬅ Represents an image map that was created **after** the round trip finished

The image maps show a snapshot of the UI. In the image maps all components that participated in the round trip are highlighted using a special border. Selecting a highlighted component results in selection of all the corresponding expressions in the expression list. On the other hand selecting an expression in the expression list loads the corresponding image map and highlights the relevant component. This relationship between the selections of the expression list and the image maps enables you to easily navigate inside the round trips.

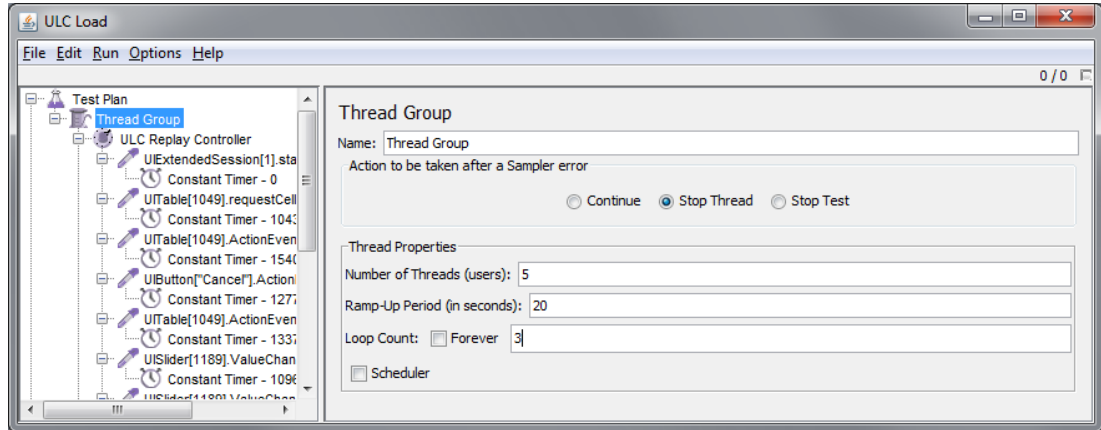


- Select **File > Save** to save the test plan.
- You can reload (**File > Open** or **Rt Clk + Open in popup menu**) the saved Test Plan and play it later provided:
 1. you use the same versions of ULC Load and JRE that were used at the time of recording the Test Plan.
 2. you have specified all the jar files containing client side customizations classes (such a client coder registry provider, custom connector provider, custom data types for which coders have been written, etc.) in the property *lax.class.path* in the file *<ULCLoad install dir>/bin/ULCLoad 3.1 UI.lax*.

How To Play a Test Plan

Follow these steps to play a test plan.

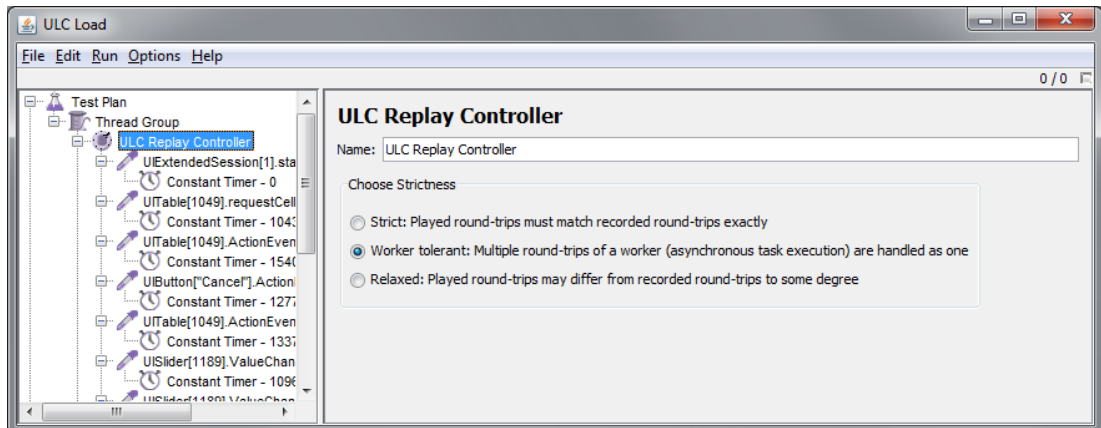
- Select and configure the **Thread Group** node. For example, change the number of threads to 5, the ramp-up period to 20, and the repeat count to 3



A **Thread Group** is the starting point of a test plan and it determines the number of threads ULC Load will use to execute the test plan. On a **Thread Group** you can set:

- **Number of threads**
Each thread simulates a concurrent connection to the ULC application to be load tested.
- **Ramp-up period**
The amount of time to start or ramp up to all threads. The interval between starting each thread is equal to *ramp-up period / number of threads*.
- **Loop Count**
The number of times a thread executes the test plan. Check the **Forever** option for looping infinitely.
- **The action to be taken after a Sampler error**
Continue: ignore error and continue with test plan.
Stop Thread: stop the thread that caused the error.
Stop Test: stop the whole test.
The recommended option is to stop the thread in case of an error.

- Choose in the **ULC Replay Controller** how strict played round-trips must match recorded round-trips.



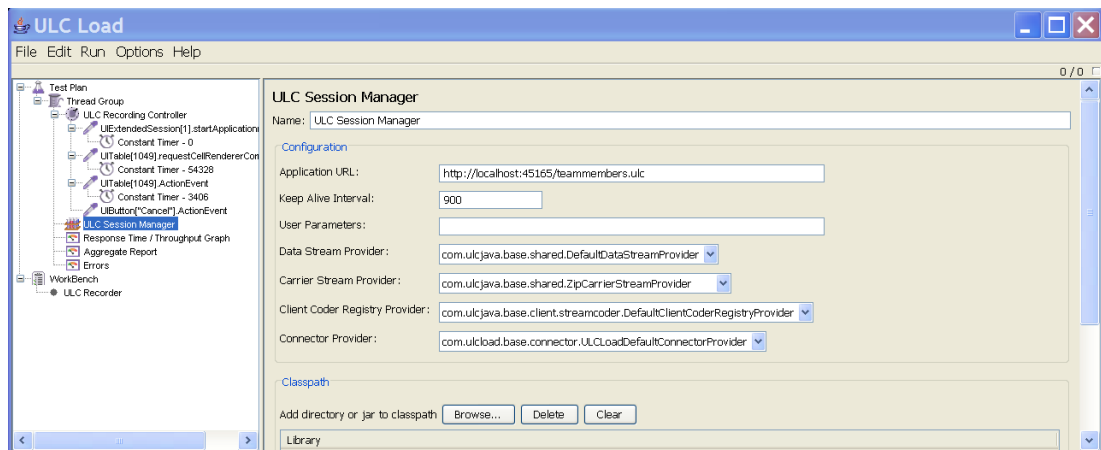
There are three levels of strictness available:

- **Strict:** the played round-trips must match the recorded round-trips exactly. Let us look at an example for illustration: if during recording a dialog was uploaded in the n-th round-trip, ULCLoad expects to receive that dialog during the test exactly at the n-th round-trip, not in an earlier and not in a later round-trip. Every violation of this condition immediately causes a sampler error (see above at **Thread Group** about possible ways to respond to a sampler error). This mode follows the "fail early" principle: as soon as the recorded and played scenarios deviate from each other, an error is thrown. For users who are familiar with ULCLoad 3.0.x, this strict mode was the one and only mode that was available in those versions.
- **Worker tolerant:** combines the strict mode with a relaxed mode as long as there are incomplete server-side workers. **ULCWorkers** in the style of *javax.swing.SwingWorker* represent in ULC long running, asynchronous server-side tasks. Such tasks typically come along with client-side polling and pose a challenge in load testing, because the number of polling round-trips varies (especially under varying load), which in turn leads to a nondeterministic application behavior. When this mode is active and the recorded scenario involves such workers, multiple round-trips, from the one starting a worker up to the one notifying the client about the worker's completion, are collapsed into a single round-trip. As this step irreversibly changes the test plan, the user is asked to save it prior to starting the test. During the test, ULCLoad introduces additional round-trips as long as uncompleted workers do exist. In summary, this mode allows long running, asynchronous server-side tasks (as long as they make use of ULC's worker abstraction), which in most cases lead to unusable test plans when applying the strict mode. Apart from workers, this mode follows the "fail early" principle just as the strict mode.
- **Relaxed:** every component in either the recorded or played round-trip, for which no counterpart (in the corresponding other round-trip) is found, is put aside in order to match it in a future round-trip. In our example that means: if during recording a dialog was uploaded in the n-th round-trip,

ULCLoad allows to receive that dialog during the test not only at the n-th round-trip, but also earlier or later. In the former case (earlier), it will be matched against the dialog's counterpart of the n-th recorded round-trip. In the latter case (later), it is matched immediately against the dialog's counterpart of the n-th recorded round-trip as that unmatched counterpart was put aside. **Warning:** This mode follows the "fail late" principle: any deviation between recorded and played scenarios is concealed, the load test is driven as far as possible while accepting the risk that a played scenario runs completely out of control. While this mode may save an otherwise unusable test plan, it comes with risks. The test may provoke otherwise impossible application states and with it, both "impossible" exceptions and even corrupt data on the backend. Needless to say that this mode should only be applied as a last resort and only if the user is aware of the possible consequences!

- Take a look at the **ULC Session Manager**. The ULC Session Manager specifies the URL the recorded test plan is executed against.

In addition, the other configuration settings of the previously recorded test plan have been copied and may be modified in order to simulate a different configuration for the test plan execution.



- The following **Listeners** have been created and added to the test plan automatically during recording.
 - **Response Time / Throughput Graph**
 - **Aggregate Report**
 - **Errors**
 - **Round-Trip Analyzer**

These listeners gather and visualize test plan execution data. For more details on the functionality of each listener, see the **How To Review Test Results** and **How To Analyze Test Issues** section.

- Select the **Run > Start** menu item to start the test plan.

The little square in the upper right corner turns green. That is how ULC Load signals that the test plan is currently running.

- Wait until the little square in the upper light corner turns gray again. This is how ULC Load signals that the test plan has finished.

In case you configured the **Thread Group** to loop forever, you will have to stop the test run manually. Use the **Run > Stop** menu item to do this.

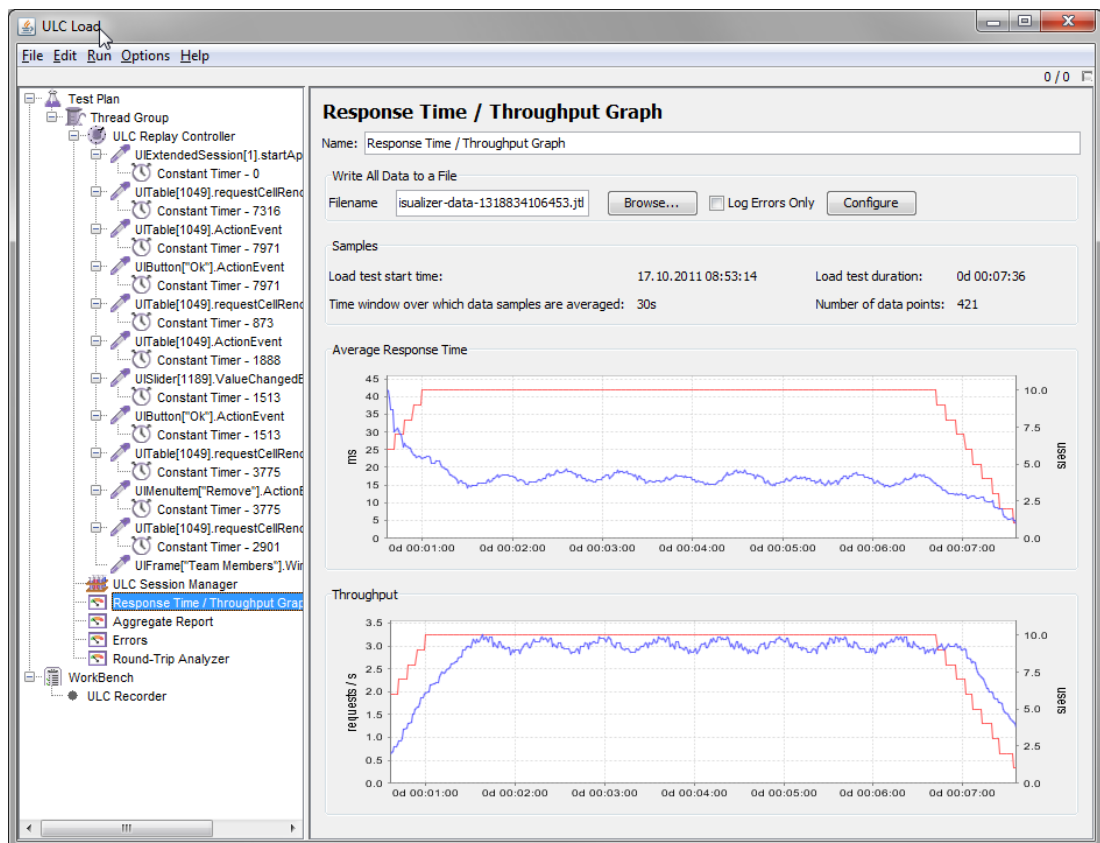
How To Review Test Results

You can review the results of the test plan execution using the information provided by the various **Listeners** configured for the test plan.

In ULC Load, listeners provide access to the information ULC Load gathers while the test plan runs. Some listeners plot the response times on a graph, while others provide aggregation information. Every listener provides an optional entry field to indicate the file name to store the collected data. When using the test plan skeleton, a file name pointing to the ULC Load data directory (ulcload-/ui/data) has been defined for every listener of the test plan.

The following listeners are a good starting point for your investigations:

- **Response Time / Throughput Graph**



The response time / throughput graph displays the average response time and the number of requests per seconds. The average response time is calculated as a moving average over a window of time. The graph also shows number of users. The graph is updated real-time during the test plan execution. In addition the following information is also displayed:

- **Load test start time:** The time when the load test started.

- **Load test duration:** The time taken to complete the load test.
- **Time window:** The time window over which the moving average of response times is calculated. At present this is fixed at 30 seconds. The moving average is calculated to be able to have a smoother graph sans outliers.
- **Number of data points:** This is the number of data points collected over the duration of the load test. Each data point is an average of response times over the previous 30 seconds. The first data point appears after 35 seconds i.e., 30 seconds for computing the average and 5 seconds to discount initialization related delays.

- **Aggregate Report**

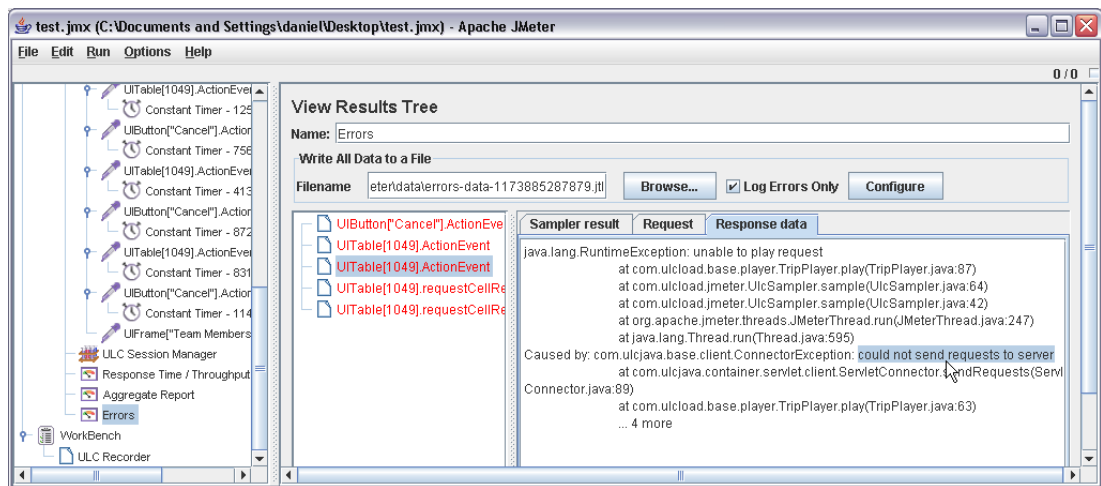
The screenshot shows the 'Aggregate Report' window in the ULC Load application. The window title is 'ULC Load' and it has a menu bar with 'File', 'Edit', 'Run', 'Options', and 'Help'. On the left, there is a 'Test Plan' tree view showing a hierarchy of components like 'Thread Group', 'ULC Replay Controller', and various UI components. The main area of the window is titled 'Aggregate Report' and contains a form for configuring the report. The form includes a 'Name' field set to 'Aggregate Report', a 'Write All Data to a File' section with a 'Filename' field (set to 'isualizer-data-1318834106468.jtl') and buttons for 'Browse...', 'Log Errors Only', and 'Configure', and an 'Export Table to a File' section with a 'Filename' field and buttons for 'Browse...' and 'Export'. Below the form is a table with the following data:

Label	# Samples	Average	Median	90% Line	Min	Max	Error %	Throughput	KB/sec
UIExtende...	100	103	101	116	92	142	0.00%	14.6/min	0.00
UITable[1...	400	10	5	29	1	49	0.00%	54.1/min	0.00
UITable[1...	200	19	29	45	2	56	0.00%	28.4/min	0.00
UIButton["...	200	3	3	6	1	10	0.00%	28.8/min	0.00
UISlider[1...	100	3	3	5	2	9	0.00%	14.6/min	0.00
UIMenuite...	100	2	2	4	2	6	0.00%	14.6/min	0.00
UIFrame["...	100	2	2	3	0	6	0.00%	14.6/min	0.00
TOTAL	1200	16	4	45	0	142	0.00%	2.7/sec	0.00

The aggregate report creates a table row for each differently named **ULC Sampler** in your test plan. The complete table can be exported as is in CSV format. For each request, it aggregates the following information:

- **# Samples**
The number of samples
- **Average**
The average time of a set of results
- **Median**
The median is the time in the middle of a set of results.
- **90% Line**
90% of the results finished faster than the line.
- **Min**
The lowest time of a set of results
- **Max**
The longest time of a set of results
- **Error %**
Percent of requests with errors

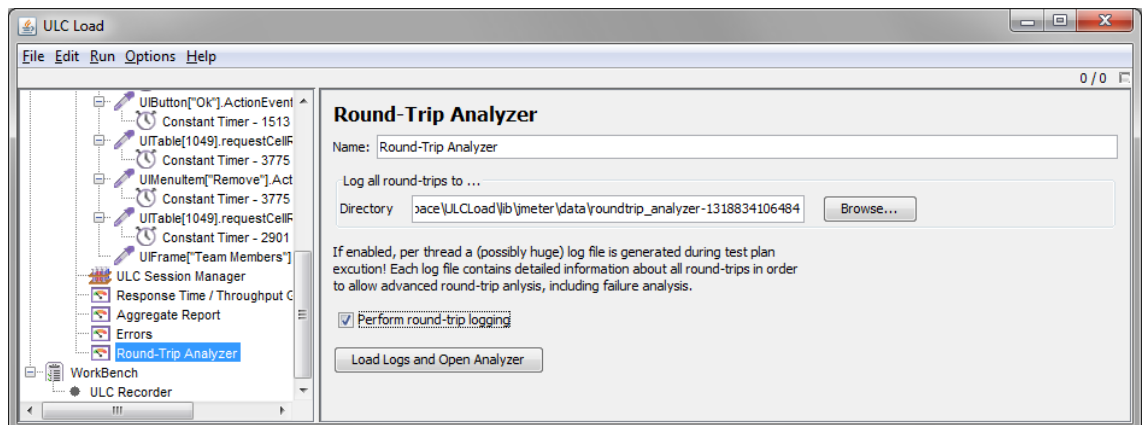
- **Throughput**
Throughput measured in requests per second/minute/hour. The throughput number represents the actual number of requests/period the server handled. This calculation includes any delays you added to your test and ULC Load own internal processing time.
- **Kb/sec**
Not used for **ULC Samplers**
- **Errors**
The View Results Tree shows a tree of all sample responses, allowing you to view the response for any sample. If a ULC Sampler fails, it places the error reason in the response data tab. You may choose to log only errors.



How To Analyze Round-Trips

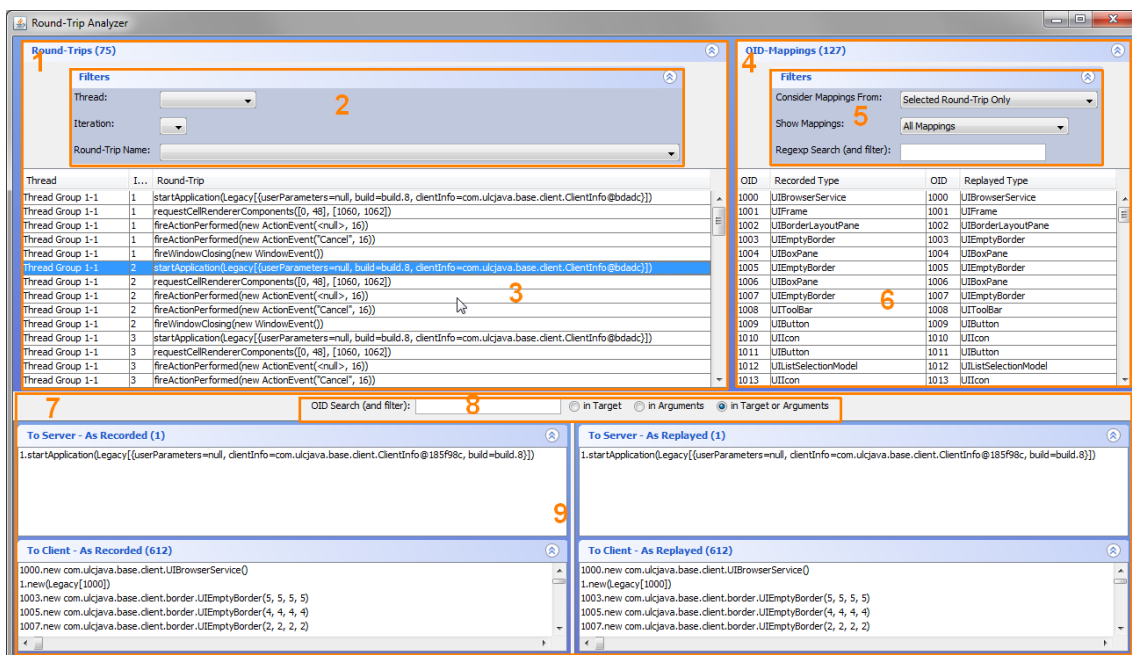
Analyzing round-trips is typically done when errors occur during the load test and the **Errors** listener (s. above) does not provide enough information to locate the cause of the error. In order to allow round-trip analysis, take the following steps:

- Round-trip logging has to be enabled on **Round-Trip Analyzer** first. Round-trip logging is disabled by default as for every thread a possibly huge log file is written during the load test. For that reason, we also recommend to enable round-trip logging only if there is a real need to analyze round-trips or associated errors!



- Execute the load test as described in **How To Play a Test Plan**.
- Load round-trip log files and open analyzer GUI on **Round-Trip Analyzer**. As the log files may be possibly huge, loading and visualizing them may both take quite some time and consume a lot of memory. You might consider moving unneeded log files first in order to speed up the process and reduce the amount of memory needed, for example if you need to analyze only the round-trips of a few threads. If you need to increase heap space, you can do so in the corresponding .lax file in `<ULCLoad Install Dir>/bin` (specify/increase the `lax.nl.java.option.additional=-Xmx...m` property). Note, that you have the possibility to analyze the already loaded log files even when cancelling the loading operation.

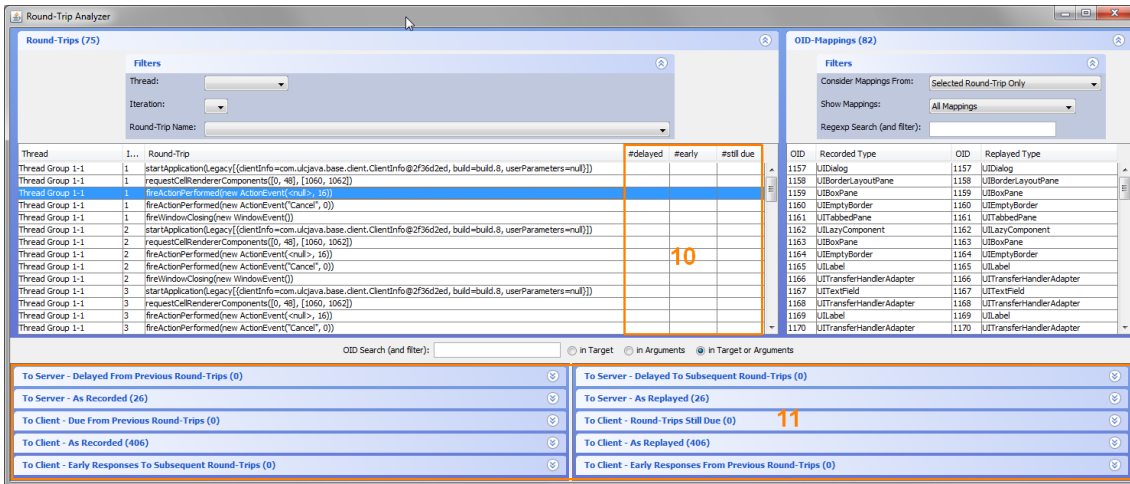
The analyzer GUI is structured as follows (see marked and labeled regions in the screen shots below):



In the top left you find the round-trips area (1) with its filtering capabilities (2) and the filtered round-trips table (3). The name of a round-trip is the same as the name of the ULC Sampler that actually performed the round-trip during load test. Without selecting a round-trip in (3), the other areas, namely (6) and (9), stay empty.

In the top right you find the OID mappings area (4), with its filtering capabilities (5) and the filtered OID mapping table (6). The OIDs are internal identifiers of ULC's user interface elements and one of ULCLoad's tasks during the load test is to identify for every recorded user interface element the corresponding replayed user interface element (their OIDs respectively). See **Architecture** for more details.

In the bottom area you find the expressions area (7), again with its filtering capabilities (8) and multiple expression lists (9). Normally to any expression found in a list to the left, a corresponding expression should be found in a list to the right. If the matching mode in **ULC Replay Controller** (see **How To Play a Test Plan**) is either *strict* or *worker tolerant*, you will find four lists as in (9): at the top those sent to the server, at the bottom those received from the server, at the left those sent/received during recording, and at the right those sent/received during the load test. However, if the matching mode is *relaxed*, you will find ten expression lists (11).



The additional expression lists hold expressions that were

- expected but not (yet) received (*To Client - Due From Previous Round-Trips* and *To Client - Round-Trips Still Due*)
- received earlier than expected (*To Client - Early Responses To Subsequent Round-Trips* and *To Client - Early Responses From Previous Round-Trips*)
- delayed because other expressions were not (yet) received (*To Server - Delayed From Previous Round-Trips* and *To Server - Delayed To Subsequent Round-Trips*).

In the case of *relaxed* matching mode, the round-trip table also contains three additional columns (10) listing the number of expressions not received, received early, or delayed respectively.

As the GUI holds possibly huge amounts of information, it is crucial to apply filters in order to find the relevant information and in general not to get lost. For that reason let us quickly go through the filtering possibilities:

- In (2) you may reduce the round-trips displayed in (3) by filtering them according to the thread (show only round-trips of a specific thread), the iteration (show only round-trips of a specific iteration), and the round-trip name (same as sampler name).
- In (5) you may extend the OID mapping entries not only to contain the (new) entries from the currently selected round-trip (for example the 15th round-trip of the third iteration of the thread *Thread Group 1-1*), but to contain all entries of the selected thread and iteration up to and including the selected round-trip (for example the round-trips 1 to 15 of the third iteration of the thread *Thread Group 1-1*). After extending the OID mapping entries like this, the complete OID mapping of a single user session is shown as it was available to ULCLoad at the moment when the selected round-trip ended.

Often only a small percentage of OID mapping entries are interesting. For example when a recorded OID is matched against either a played OID of a different value or against multiple played OIDs. OID mapping entries where the corresponding user interface component types are different are interesting too.

Last but not least, you may enter a regular expression that has to be found in any displayed value of the OID mapping entries. That way you may search for

specific types (e.g. *UIFrame* or *UI.*Border*) or specific OIDs (e.g. *1011* or *10ld1*).

- In (8) you may filter the expression lists in order to show only those affecting specific OIDs. OIDs need to be separated by commas and any invalid OID is simply ignored, e.g. *1001* (capital 'o' instead of digit '0') or *1001m* (typing error).

Lastly, the analyzer GUI supports quick navigation from an OID mapping entry to the expressions creating the corresponding user interface components. Just select the context menu entry **Go To Creating Expression** on an OID mapping entry of your choice. If necessary, the corresponding round-trip is selected and the expression lists containing the creating expressions are expanded.

How To Parameterize a Test Plan

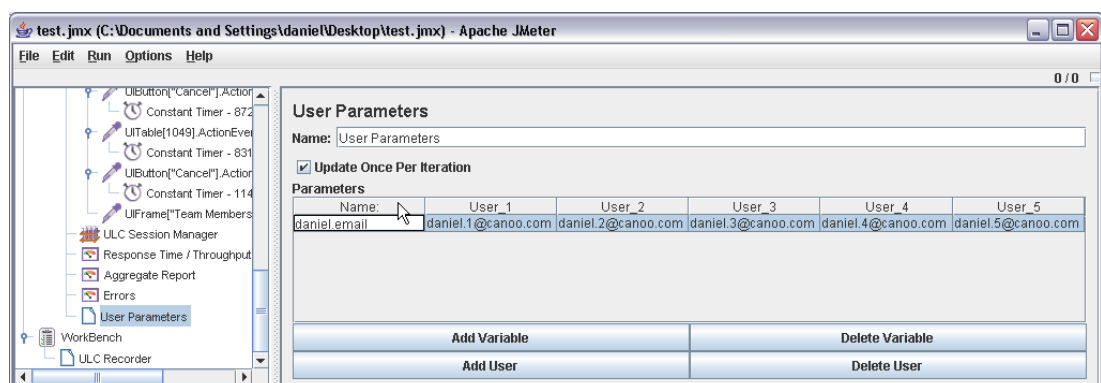
ULC Load allows parameterization of a Test Plan. The primary reason to do this is to provide each virtual user with a separate application context while running the test plan. A typical example of how to achieve this is to parameterize the *username* and *password* in the login dialog of an application.

In the following example, we parameterize the employee email address name:

- Use the context menu to add a new **Pre Processors > User Parameters** node to the existing **Thread Group** tree node.

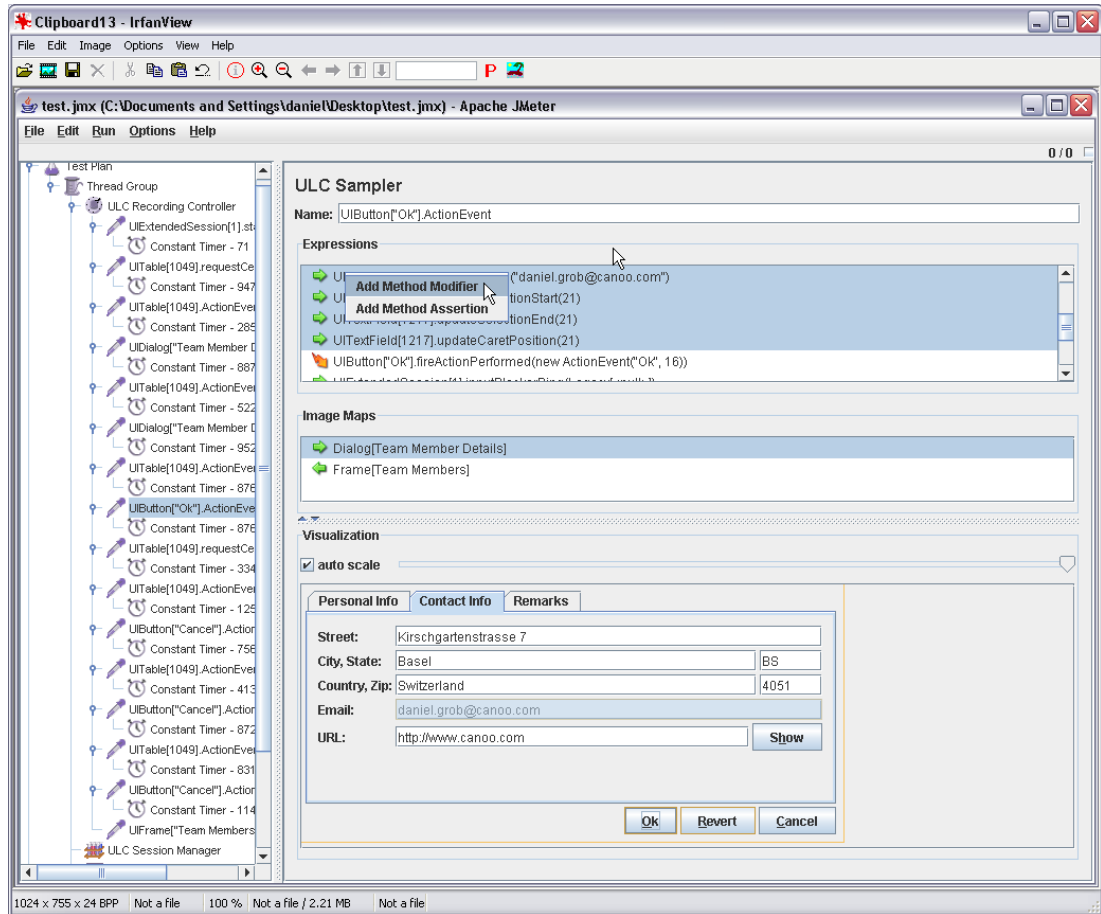
The **User Parameters** node enables you to define a parameter that has separate values per virtual user.

- Click the **Add Variable** button to create a new parameter and type *daniel.email* as parameter name into the corresponding table cell.
- Click the **Add User** button several times to create value cells for each virtual user and type the virtual user dependant values for the *daniel.email* parameter into the corresponding table cells.



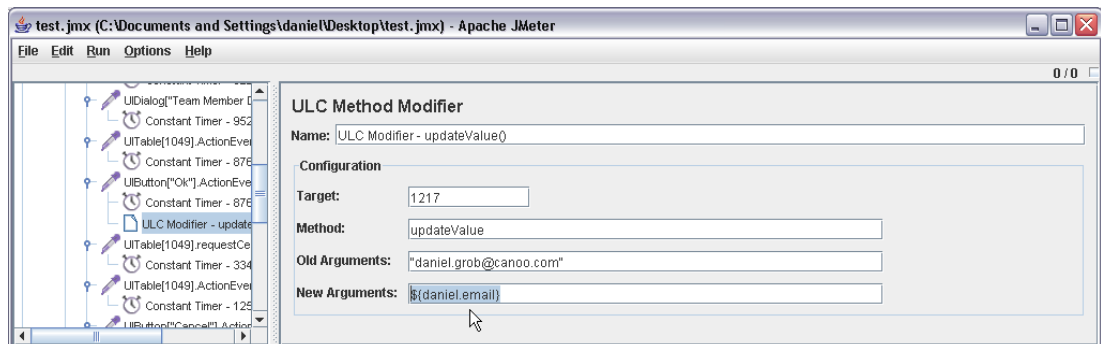
- Select the ULC Sampler in the test plan where you recorded the change in the employees email address.
- Select the image map for the detail dialog that was created before the roundtrip started. This loads the image map into the visualization pane at the bottom.
- Select the *Email* text field in the image map. This selects the expressions that are related to this text field in the expression list above.

- Right click on the `UITextField.updateValue(...)` Expression and choose **Add Method Modifier** from the context menu.



- Expand the corresponding **ULC Sampler** and select the just created **ULC Modifier** node. Type in the name of the user parameter enclosed in braces into the *New Arguments* text field, i.e. `#{daniel.email}`.

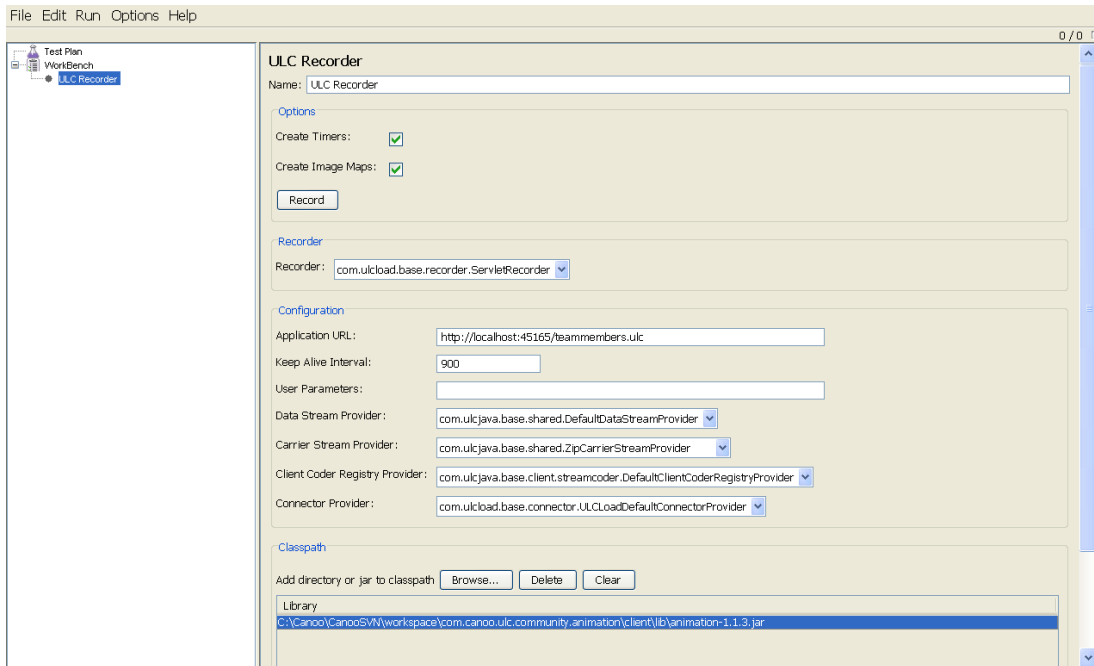
The braces notation has the effect that the enclosed value evaluates to the corresponding parameter value. Now when you execute the test plan, the method modifier will be applied before sending the request, changing the content of the *Email* text field to the corresponding virtual user dependant value.



How To Handle Extensions

In case your ULC application is using extension widgets, you need to specify the jar file containing the client side proxies of your extended ULC widgets in the class path of

ULC Load. Specifying the class path in the ULC Recorder does this. For example, in the following picture we are setting the *JGoodies Animation* extension's client side jar on the recorder path.



How To Handle Custom Launchers

What a launcher is to ULC, a recorder is to ULC Load. If you have a custom launcher for your ULC application and if you want to have the same parameters (such as Look & Feel, custom services, etc.) as the custom launcher while you are recording a scenario, then you need to create a custom recorder for your application.

Your custom recorder should subclass *com.ulcload.base.recorder.ServletRecorder*. This class is available in the *ulcload.jar* library in the *base/lib* directory of your ULC Load installation. In addition, there are source file stubs available in the *base/src* directory for enabling code completion in your preferred IDE.

You then can override the following methods:

- **createInitializer()**
Factory method to create a custom initializer.

Copy all environment initialization code from your launcher to your custom recorder:

```
public class CustomRecorder extends ServletRecorder {
    @Override
    protected Runnable createInitializer() {
        return new Runnable() {
            public void run() {
                try {
                    String laf =
                        UIManager.getSystemLookAndFeelClassName();
                    UIManager.setLookAndFeel(laf);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        };
    }
}
```

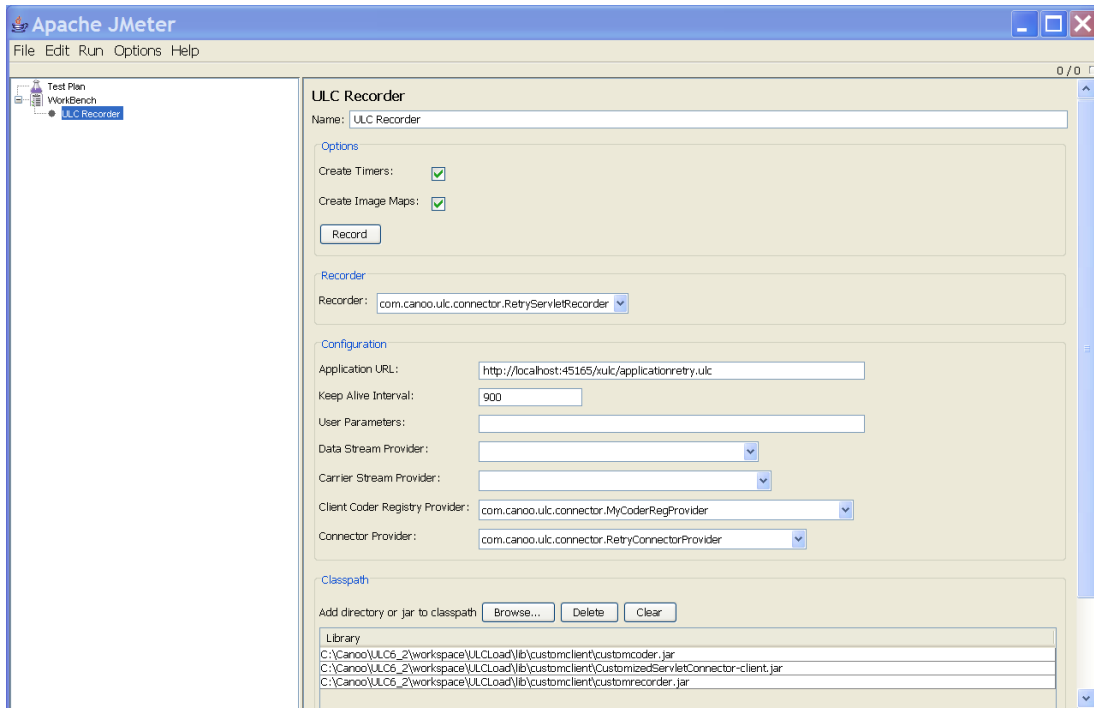
- **createBrowserService()**
Factory method to create a custom browser service.
- **createFileService()**
Factory method to create a custom file service.
- **createMessageService()**
Factory method to create a custom message service.
- **createRequestPropertyStore()**
Factory method to create a custom request property store.
- **createSession()**
Factory method to create a custom client-side ULC session.

Please see ULC documentation for details on the above objects that are created by the factory methods.

Put the jar file containing the custom recorder class in the path of the **ULC Recorder** in ULC Load. Then you can choose your custom recorder class as the value for the **Recorder** field of **ULC Recorder**.

To customize the *com.ulcjava.base.client.IConnector* implementation to be used by the recorder and the player, provide a custom implementation of *com.ulcload.base.connector.IConnectorProvider*. By default, ULC Load uses *com.ulcload.base.connector.ULCLoadDefaultConnectorProvider* which provides ULC's *com.ulcjava.container.servlet.client.ServletConnector*. In the custom connector provider, implement the *getConnector()* method to return an implementation of *IConnector*. Add the jar file containing custom *IConnectorProvider* to the path of the **ULC Recorder** in ULC Load. The custom connector provider will then be available for selection in the **Connector Provider** combobox.

The following picture shows how to specify a customized recorder, client coder registry provider and connector provider classes. The jar files containing these classes are specified under **Classpath**.



How To Run a Distributed Load Test

Note: this option is only available for ULC Load Enterprise licenses.

The amount of load able to generate on a ULC application depends on the power of the load driver(s) involved. In contrast to a standard ULC Load license, a ULC Load Enterprise license is not limited to one load driver, but allows for running unlimited number of load drivers on unlimited number of load driver machines.

A distributed load testing environment distinguishes between the master node and one or more slave nodes. The master node runs the ULC Load UI whereas each slave node runs one ULC Load slave driver. In a distributed test, the ULC Load UI starts and stops the slave drivers and collects all test results.

The ULC Load slave drivers don't have a user interface, their only purpose is to generate load on the ULC application on behalf of the ULC Load UI. They are completely controlled by the ULC Load UI.

For each slave node, follow these steps to install and start the ULC Load slave:

- Install ULC Load.
- Open the **Start** or **Program** menu and select **ULC Load 3.1 > ULC Load Slave Driver** to start the ULC Load slave driver. This opens an empty console window.
- Discover the slave node's IP address.
- If the Test Plan to be played was recorded with jar files containing client side customizations such as custom client coder registry provider, custom connector

provider, etc. (see section How to Handle Custom Launchers), then you need to:

1. copy the jar files containing custom classes to each slave node.
2. specify the jar files' complete path in the property *lax.class.path* in the file *<ULCLoad Install Dir>/bin/ ULC Load 3.1 Slave Driver.lax* before starting ULC Load Slave driver on that slave node.

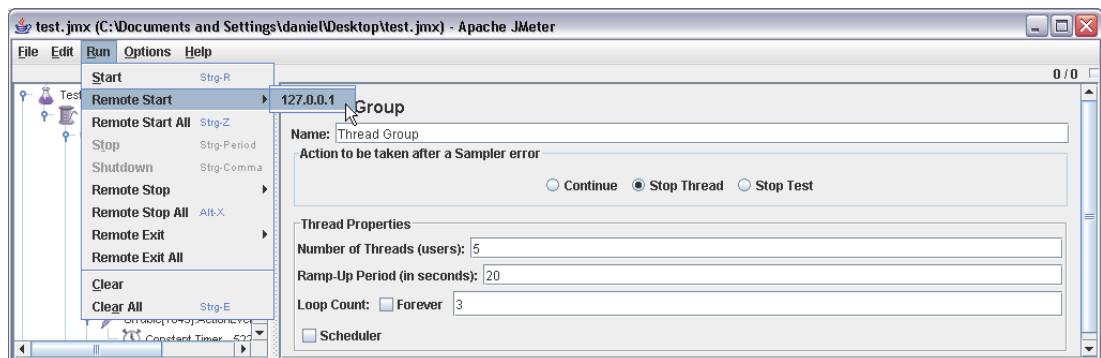
Configure the available slave nodes on your master node:

- Open the **Start** or **Program** menu and select **ULC Load 3.1 > ULC Load 3.1 Directory** to open ULC Load's directory.
- Open the file *<ULC Load Directory>/ui/bin/jmeter.properties*
- Replace the current value of the **remote_hosts** entry with the comma-separated list of the IP addresses of your slave nodes. The **remote_hosts** entry configures the available slave nodes.
- Save the customized *<ULC Load Directory>/ui/bin/jmeter.properties* file.

Control the slave nodes from the ULC Load user interface running on the master node:

- Open the **Start** or **Program** menu and select **ULC Load 3.1 > ULC Load 3.1 UI** to start the ULC Load user interface.

The **Run > Remote Start** menu should now list the IP addresses of the configured slave nodes.



- Record a new or open an existing test plan.
- Select the **Run > Remote Start All** menu item to start the test plan on all slave nodes or select the **Run > Remote Start > IP Address** to start the test plan a specific slave node.

When using the **Forever** option in the **Thread Group** you stop the test run with help of the **Run > Remote Stop All** or the **Run > Remote Stop > IP Address** menu item.

How To Use Client Certificates

When load testing an application over SSL and concurrent user sessions need to relate to different users (for example because the application ensures that a user is not logged in twice), multiple client certificates need to be used. Client certificates are configured in the configuration file *clientCertificateConfig.xml*. During recording, ULCLoad uses the (first) client certificate specified under the XPath

configure/recorder/certificate. During the load test, ULCLoad uses the client certificates under configure/player/certificate, thereby using the first player certificate for the first thread, the second player certificate for the second thread, and so on. So, in order to use client certificates, take the following steps:

- Create a new xml file named clientCertificateConfig.xml in the directory <ULCLoad installation directory>/ui/bin.
- Edit clientCertificateConfig.xml according to your needs. Here is a sample clientCertificateConfig.xml:

```
<configure>
  <!--
    client certificate for recorder (only the first is used)
  -->
  <recorder>
    <certificate>
      <file>client1.p12</file>
      <password>changeit</password>
      <type>PKCS12</type>
    </certificate>
  </recorder>

  <!--
    list of client certificates for players (many are allowed)
  -->
  <player>
    <certificate>
      <file>client2.p12</file>
      <password>changeit</password>
      <type>PKCS12</type>
    </certificate>
    <certificate>
      <file>client3.p12</file>
      <password>changeit</password>
      <type>PKCS12</type>
    </certificate>
  </player>
</configure>
```

- In **ULC Recorder** choose `com.ulcload.base.connector.ULCLoadSSLConnectorProvider` as a connector provider.

Best Practices

Add Metadata to your ULC Application

You can help ULC Load to improve the readability of the scenario. You can do so by adding metadata to the component in your ULC application. ULC Load uses this metadata to provide better descriptions for the expressions that form a scenario. For more information about expressions see section *Recording* in the *Architecture* chapter.

ULC Load evaluates the following metadata:

- **name** property of a **Component**
- **labelFor** property of a **JComponent**
- **toolTipText** property of a **JComponent**

Therefore, as far as possible, specify the above attributes on your ULC Widgets.

Record Robust Scenarios

While recording a scenario, keep in mind that the scenario will be played concurrently during a test run. This can lead to race conditions:

- In your scenario you delete item *foo*
Virtual user *A* deletes item *foo*
Right after that virtual user *B* deletes item *foo* as well
→ Error because item *foo* is not available anymore
- In your scenario you delete item *foo*
Virtual user *A* deletes item *foo*
In the next iteration the same virtual user deletes item *foo* again
→ Error because item *foo* is not available anymore

Therefore try to respect the following rules of thumb in your scenarios:

- Don't change shared state
Shared state is state that can be read by different virtual users.
- Restore changed private state at the end of the scenario
Private state is state than can be read only by one virtual user.

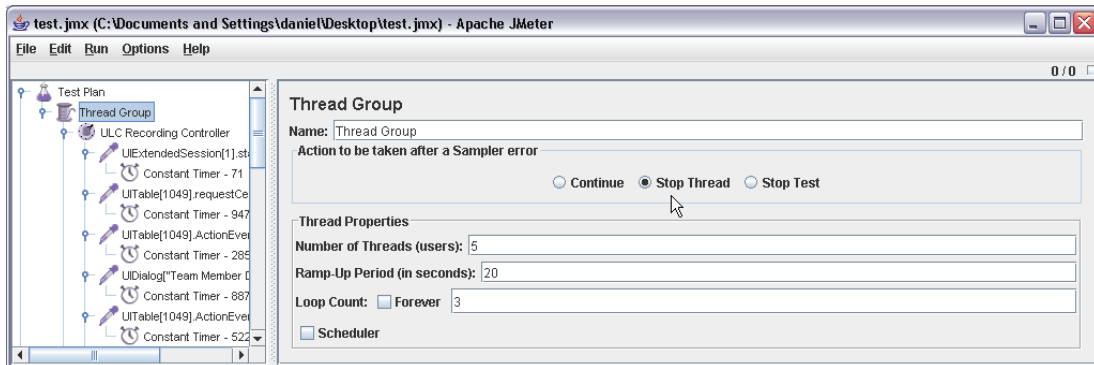
Play what you Record

Use parameterization judiciously. ULC Load is a load test tool and not a functional test tool.

Failed Samplers are Serious

Your application is in an inconsistent state if a ULC Sampler fails. In such a case don't continue with the scenario, just stop the virtual user.

You can configure the action to be taken on a sampler error in the Thread Group node. Just set the **Action to be taken after a Sampler error** property to the value **Stop Thread**.



Ramp Up Slow

Each virtual user should be able to play the scenario several times before the next virtual user is started in the test run. This ensures that the **Throughput** (ULC Samplers per time) stabilizes itself before the next virtual user starts and increases the **Throughput** again.

As a result you should see a discrete **Throughput** graph that makes a jump whenever a new virtual user is started. Typically the jumps get smaller and smaller until at some point in the time the jumps disappear completely. The point where the jumps disappear is the point where your ULC application is not able to consume any more load. From then on additional virtual user just increase the response time for all active virtual users.

Let the Virtual Users Think

We recommend that you add think time to your scenario by adding at least one **Timer** to your **Thread Group**. If you do not add think time, ULC Load could overwhelm your server by making too many requests in a very short amount of time.

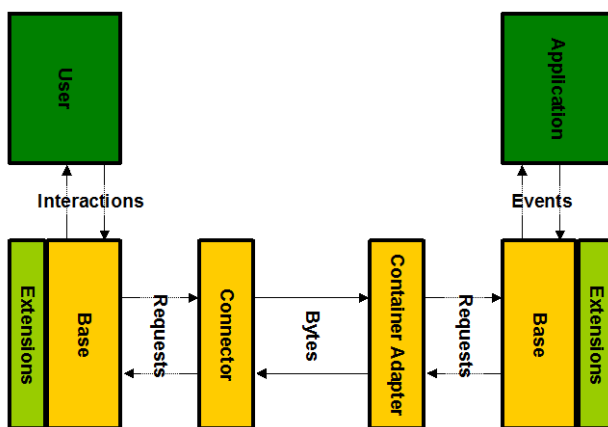
Architecture

Recording

Before we go into the details of how ULC Load records a load scenario we first describe how ULC exchanges information between the user and the ULC application.

The ULC client recognizes all relevant user interactions. For each recognized user interaction ULC sends a corresponding request from the client to the server using the network. On the server side the requests are transformed into events and are then processed by the ULC application. As a result of this processing the altered application state is sent back from the server to the client.

The following diagram describes the ULC modules involved in request generation, transmission, and processing:

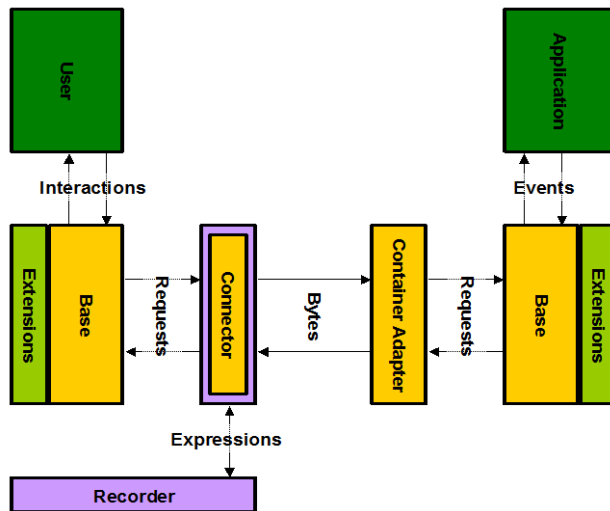


From this description we can see that the user scenario can be expressed as a series of items such as:

- User interactions in the application layer
- Requests in the communication layer
- Bytes in the network layer

ULC Load works with the items at the communication layer level to form a user scenario. Thus the recorded user scenario is a series of requests and corresponding responses. Since the request format that ULC uses on the communication layer was not designed to be human readable, ULC Load always converts requests into a human readable format called expressions.

The following diagram shows how the ULC Load recorder integrates into the ULC modules:



Each expression consists of a message to a user interface element plus optional message arguments. Each user interface element is identified by a unique id. Examples for expressions are:

- **1205.setValue("Hello World")**
(where 1205 is the reference to a text field)
- **1175.setSize(800, 600)**
(where 1175 is the reference to a window)
- **2380.actionPerformed("OK")**
(where 2380 is the reference to a button)

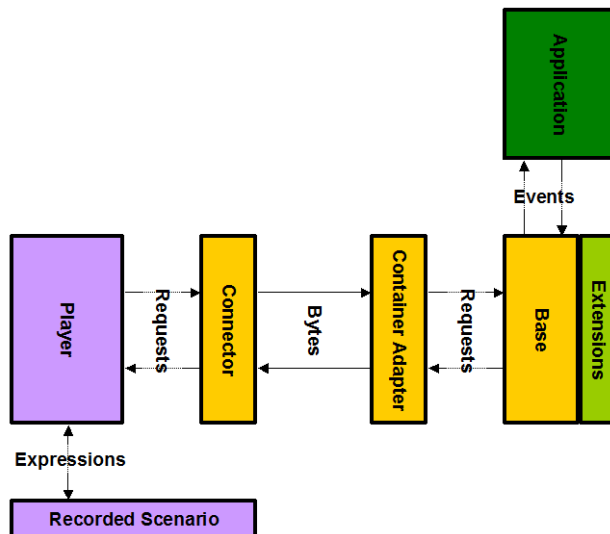
To make the expressions more readable, ULC Load gathers some meta data during the recording phase. With the help of this meta data user interface elements can be described with words and / or pictures instead of just an unique id. The previous examples can then be expressed as follows:

- **ULCTextField["Output"].setValue("Hello World")**
- **ULCFrame["Main Frame"].setSize(800, 600)**
- **ULCButton["OK"].actionPerformed("OK")**

Replaying

As described in the previous section, the recorded scenario is based on the communication layer. Therefore to be able to replay a scenario one first has to establish a connection on the communication layer. After that all recorded round trips are replayed one by one. The connection can be closed after all round trips have been replayed.

The following diagram describes how the ULC Load player integrates into the ULC modules:



With ULC Load you are able to parameterize a recorded scenario. This parameterization might have side effects on the unmodified parts of the recorded scenario:

- **Text Components**

In the recorded scenario you enter “Alice” into a text field and then set the caret to the end of the entered text. The caret index is therefore 5.

You now parameterize the scenario so that “Bob” is entered into the same text field. The caret index is now 3 instead of 5.

→ To ensure constraints on the parameterized scenario, the caret index should be parameterized as well!

- **Lazy Loading**

In the recorded scenario you enter “Alice” into a search field. After you click the search button the result table contains 10 rows and therefore 10 rows are lazily fetched from the server.

You now parameterize the scenario so that “Bob” is entered into the same search field. The result table now contains only 5 rows instead of 10.

→ To ensure constraints on the parameterized scenario, the lazy loading request should be parameterized as well!

ULC Load has a built-in mechanism to ensure constraints on the parameterized scenario. Currently the two cases described above are supported out of the box. Other constraints can be implemented as extensions of the player.

Another side effect of a parameterized scenario is that the ids of user interface elements might change, e.g. the recorded user interface element 1203 might become the replayed user interface element 1205. To respect this side effect ULC Load has a built-in mechanism to match recorded user interface elements to replayed ones. The matching process that is provided out of the box is rather restrictive and relies on the creation order of user interface elements. This means that in the replayed scenario it is not allowed to have additional user interface elements or to leave out recorded ones.

(Except if the matching mode in **ULC Replay Controller** is set to *relaxed*, see **How To Play a Test Plan** for more details on matching modes.)

All this results in the following steps while replaying a parameterized scenario:

1. Establish connection on communication layer
2. For each recorded round trip do the following
 - a. Parameterize the request
 - b. Ensure constraints on the request
 - c. Match recorded user interface elements to replayed ones
 - d. Send requests to server, receive responses from server
 - e. Consume response to support subsequent matches
 - f. Match replayed user interface elements to recorded ones
 - g. Consume response to support subsequent constraints
 - h. Perform assertions on the response
3. Close connection