ACS Performance Test

User's Documentation

Document Owner:	*Anze Zagar*
Status:	*Draft*
Availability:	*eso:/doc/DOC-ACS_Performance_Test.xml*
Creation:	*2004-03-12 (Anze Zagar)*
Last Modification:	*2004-08-13 (Steve Harrington)*

Scope

This document gives instructions on how to execute and extend the performance test suite. Also, it gives a report on the results of some of these ACS stress and performance tests, which were performed at CosyLab.

Audience

Anyone interested in ACS performance testing, in particular ACS developers and prospective users.

Table of Contents

How to Read this Document

This document's meta-information (authors, revision history, table of contents, ...) can be found above. What follows below is the body of the document. The body is composed of several sections, which may be further composed of subsections.

Typographical styles are used to denote entities of different kinds. For a full list of entities and their respective typographic conventions, please refer to the Styles section of the XML Documentation document.

When viewing the document in a non-printed form, it is possible to submit comments regarding a given section to the document's owner. This is achieved by clicking the mail icon next to the section title. For this to work, your mail must be configured to properly handle the `mailto` URLs.

1. Introduction

The ACS performance evaluation suite consists of the following kinds of tests:

1. **Startup tests:** gathering data on the startup time of individual components.
2. **Throughput tests:** finding the throughput of individual operations.
3. **Scalability tests:** determining the limits where the server can still keep up with the client requests and remain stable.

To facilitate multiple-client stress testing, Apache JMeter application [3] was used. JMeter allows

for creating automatically executable *test plans*. Additional ACS-specific plug-ins for JMeter were developed.

The tests were performed on a simple home environment and involved three Linux desktops, one of which was used as a standalone ACS server runnig only jmanager and one C++ container. All components used for these tests were bulit in C++ whereas clients were all Java made.

Analysis of the data collected by jMeter was later made manually in OpenOffice.org Calc.

1.1. Testing Computers

Computer	Processor	Memory	Operating System	Other Information
Thor/192.168.1.4	Intel Celeron, 1.8GHz, 128kB cache	768MB RAM	Fedora Core 1 (kernel 2.6.2)	• ACS Server (ACS-3.1.pre-20040107) • J2RE SE 1.4.2_03-b02 • Running in text mode • Primary client
Odin/192.168.1.5	Intel Pentium 4, 2.5GHz, 512kB cache	1GB RAM	Debian Linux (kernel 2.6.3)	• J2RE SE 1.4.2_02-b03 • Running in X, ICEWM • Secondary Client
Heimdall/192.168.1.1	AMD Athlon, 850MHz, 256kB cache	768MB RAM	Debian Linux (kernel 2.4.22)	• J2RE SE 1.4.2_02-b03 • Controlled remotely via SSH • Home server, routher, firewall

During testing there was also another Windows 98 computer connected to the network.

2. Tests

The following tests are implemented:

1. Startup tests:
    1. CORBA initialization:
        1. **TEST_1_1_1**: Server [analyze **acsStart** logs]
        2. **TEST_1_1_2**: Client [do `ComponentClient::initCorba()`]
    2. Device activation on server startup:
        1. **TEST_1_2_1**: Export/activate N={1,2,4,8,16} simple devices [prepare CDB and analyze **acsStartContainer** logs]
        2. **TEST_1_2_2**: Export/activate N={1,2,4,8,16} complex devices [prepare CDB and analyze **acsStartContainer** logs]
        3. **TEST_1_2_3**: Stop container with N={1,2,4,8,16} simple devices activated [analyze **acsStartContainer** logs]
        4. **TEST_1_2_4**: Stop container with N={1,2,4,8,16} complex devices activated [analyze **acsStartContainer** logs]
    3. Device activation from client:
        1. **TEST_1_3_1**: Connect to N={1,2,4,8,16} simple *not yet activated* devices [do `getComponent()` for each]
        2. **TEST_1_3_2**: Connect to N={1,2,4,8,16} complex *not yet activated* devices [do `getComponent()` for each]
        3. **TEST_1_3_3**: Time for client to release all N={1,2,4,8,16} simple devices [do `releaseComponent()` for each]
        4. **TEST_1_3_4**: Time for client to release all N={1,2,4,8,16} complex devices [do `releaseComponent()` for each]
2. Throughput tests (measuring operations per second = 1 / time of one operation):
    1. Synchronous operations (N={1,2,4,8,16} clients distrubited on M={1,2,4} machines):
        1. **TEST_2_1_1**: `getProperty()`
        2. **TEST_2_1_2**: `setProperty()`
        3. **TEST_2_1_3**: `Device::getCharacteristic()`
        4. **TEST_2_1_4**: `Property::getCharacteristic()`
        5. **TEST_2_1_5**: `action()`
    2. Asynchronous operations (N={1,2,4,8,16} clients distrubited on M={1,2,4} machines, callbacks are empty):
        1. **TEST_2_2_1_1**: `getProperty()` [only send requests]
        2. **TEST_2_2_1_2**: `getProperty()` [wait until requests are answered]
        3. **TEST_2_2_2_1**: `setProperty()` [only send requests]
        4. **TEST_2_2_2_2**: `setProperty()` [wait until requests are answered]
        5. **TEST_2_2_3_1**: `action()` [only send requests]
        6. **TEST_2_2_3_2**: `action()` [wait until requests are answered]
    3. Monitor tests (N={1,2,4,8,16} clients distrubited on M={1,2,4} machines, callbacks are empty):

    1. **TEST_2_3_1**: Monitor timer exactness [callbacks should be generated once per second and the exact time between two is measured]
    2. **TEST_2_3_2**: Callback throughput [set monitors to generate callbacks as fast as posible]
    3. **TEST_2_3_3**: Time for client to create a monitor for one property [`create_monitor()`]
4. Logging service performance (without IO/console output):
    1. **TEST_2_4_1**: Server logging [call synhronous action on server component to generate N={1,2,4,8,16} M={100,1k,10k,100k}-byte logs]
    2. **TEST_2_4_2**: Client logging [have client generate N={1,2,4,8,16} M={100,1k,10k,100k}-byte logs]
5. **TEST_2_5**: Error handling performance on server [have synchronous action on component call some recursive function which will in its N={1,2,4,8,16}-th iteration raise an `ACSErr` exception]
6. **TEST_2_6**: Retrieval of COB reference [N={1,2,4,8,16} clients distributed on M={1,2,4} machines should be constantly (T->Inf) requesting the reference of *already activated* device]

2.1. Notes

- Often a single operation is too fast to be exactly measurable (i.e., takes a very short, hardly measurable time to preform). Therefore, we rather measure a whole set of (e.g., 100) operations. This applies to 2.1, 2.2, 2.3.2, 2.3.3, 2.4, 2.5, 2.6 tests. The larger set we take, the better reslults we will get.
- Tests 1.3.* proved out to be causing a major memory leak on Container and had to be performed sepearately.
- Test 2.3.2 was only run in singlethreaded mode because with more threads (probably some kind of sharing violation) it was crashing Container with the following output:

```
2004-03-06T17:19:42.605 Failed to activate CORBA object
2004-03-06T17:19:42.605 (7528|81926) EXCEPTION, CORBA exception
caught
user exception, ID
'IDL:omg.org/PortableServer/POA/ObjectAlreadyActive:1.0'

2004-03-06T17:19:42.606 Failed to activate CORBA object
'SBC00:property_monitor0'
/alma/ACS-3.1/ACSSW/bin/acsStartContainer: line 156:  7528 Killed
maciContainer $COMMANDLINE
```

- Tests 2.3.1 and 2.3.3 were only performed in singlethreaded mode for no particular reason :) (did not have the time)

- Tests 2.4.* were not performed at all (did not have the time)
- All multithreaded tests were only performed on (N,M)={(1,1),(2,1),(4,1),(8,1),(1,2),(2,2),(4,2)}, where N is the number of threads and M is the number of computers.

3. Preparation of tests

3.1. ACS components/devices

I had to create three components:

SimpleBACIComponent (C++, BACI):

- 1 RWlong property with 13 default characteristics,
- 1 asyncronous action,
- 1 characteristic.

ComplexBACIComponent (C++, BACI):

- 16 RWlong properties with 13 default + 51 additional characteristics,
- 16 asyncronous actions,
- 64 characteristics.

LogErrTestComponent (C++):

- synchronous method to generate specified amount of logs on container,
- synchronous method to raise Exception with stack of given level.

I also created two simple (non-BACI) components, one in C++ and one in Java for experimenting.

I leared everything from acsexmplHelloWorld, acsexamplPowerSupply, BACI_Device_Server_Programming_Tutorial.doc and jcontexmpl's HelloDemo.

3.2. Pereparation of CDB

These components crash container if their CDB configuration files are invalid or do not exist.

I also have not found any good documentation on how to create schemas, xmls and there was nothing (not even a sample) on how to define device and property characteristics. For example, from where was I to know that `sequence` element must come in front of `attribute` elements of `complexType`.

3.3. ComponentClients

I created two clients:

- **BCTClient** that performs all the tests that require **SimpleBACIComponent** and

ComplexBACIComponent,

- **LECTClient** that performs all the logging and error handling tests (2.4.*).

At first I extended all my clients from **ComponentClient** class as in **HelloDemoClient** of **jcontexmpl**, causing that each test had to individually initialize Corba and connect to manager as also do its own finalization and disconnection. This led to some difficulties after *exactly* fifth test/client did its **tearDown()** - any further communication with manager was disabled until either process running the tests/clients (e.g. Eclipse or jMeter) or Manager was restarted.

After discussion with Klemen, we created singleton pattern for **ComponentClient**. All **ComponentClient** initialization and finalization was after then done only once per lifecycle of the entire set of tests.

3.4. jMeter clients

jMeter clients need to implement **JavaSamplerClient** interface. I created the following:

- **CCSampler**: only used for initialization and finalization of singleton **ComponentClient** object. This should be done at the beginning and at the end of the entire test plan.
- **T112Sampler**: performs 1.1.2 test and takes parameters:
  - **ThreadNum** - identifier of the thread to be used in sample's label (e.g. **${__threadNum}**),
- **T13Sampler**: performs 1.3.* tests and takes parameters:
  - **ThreadNum** - identifier of the thread to be used in sample's label (e.g. **${__threadNum}**)
  - **NoOfDevices** - Number of COB devices [1-16] needed to be activated/deactivate (e.g. {1,2,4,8,16})
  - **TEST_1_3_** - Identifies which test we want to perform - the remaining part of its name (in this case one of {1,2,3,4})
- **T21Sampler**: performs 2.1.* tests and takes parameters:
  - **ThreadNum** - identifier of the thread to be used in sample's label (e.g. **${__threadNum}**)
  - **SizeOfSet** - Size of the set - number of operations to be made together in one test
  - **"TEST_2_1_** - Identifies which test we want to perform - the remaining part of its name (in this case one of {1,2,3,4,5})
- **T22Sampler**: performs 2.2.*.* tests and takes parameters:
  - **ThreadNum** - identifier of the thread to be used in sample's label (e.g. **${__threadNum}**)
  - **SizeOfSet** - Size of the set - number of operations to be made together in one test
  - **Delay** - Delay time between sending two sequent requests in the set (in milliseconds) [maybe this should be extended to nanoseconds]

    - **TEST_2_2_** - Identifies which test we want to perform - the remaining part of its name (in this case one of {1_1,1_2,2_1,2_2,3_1,3_2})
- **T23Sampler**: performs 2.3.* tests and takes parameters:
    - **ThreadNum** - identifier of the thread to be used in sample's label (e.g. **${__threadNum}**)
    - **SizeOfSet** - Size of the set:
        - for 2.3.1: not used
        - for 2.3.2: number of monitor's callbacks to wait for until ending the test
        - for 2.3.3: number of monitors to create together in one test
    - **Delay** - only for 2.3.2 and provides the time of monitor callback timer (in 100ns)
    - **TEST_2_3_** - Identifies which test we want to perform - the remaining part of its name (in this case one of {1,2,3})
- **T24Sampler**: performs 2.4.* and 2.5 tests and takes parameters:
    - **ThreadNum** - identifier of the thread to be used in sample's label (e.g. **${__threadNum}**)
    - **LogCount** - for 2.4.* only and provides the number of logs to be generated
    - **LogSize** - for 2.4.* only and provides the size of each log
    - **ErrorIterations** - for 2.5 only and provides the number of levels of expection stack
    - **TEST_2_** - Identifies which test we want to perform - the remaining part of its name (in this case one of {4_1,4_2,5})
- **T26Sampler**: performs 2.6 test and takes parameters:
    - **ThreadNum** - identifier of the thread to be used in sample's label (e.g. **${__threadNum}**)
    - **SizeOfSet** - Size of the set - number of COB retrievals to be made together in one test

3.5. Preparation of jMeter test plan

All tests were added to the plan and their parameters chosen *wisely* (correlated with **TESTS_SIZE** parameter). Test plan can be further adjusted through some parameters:

- **MANAGER** (corbaloc of the manager)
- **DISTSYNC_SERVER** (location of DistSync server)
- **REMOTE_COUNT** (number of computers that will be running the tests)
- **THREADS_COUNT** (number of threads per computer that will be running the tests)
- **TESTS_SIZE** (allows adjusting the size of tests)

3.6. DistSync

Because in case of more clients (number of clients equals threads times machines) we want all clients to perform each test simultaneously (starting together and waiting for the others until they are all done, so that they will all simultaneously proceed with the next test), we have to be

synchronizing them before each test. To do this Klemen created distsync client for jMeter that connects to distsync server (run as a service in another process or even on a remote computer) and waits to be notified when all the other clients get connected.

This is how it works: more clients are running our jMeter test plan and at some point they all come to one (most commonly the same) distsync sampler client element in the plan. This element will connect them to the distsync server which will then delay each client until the number of all connected clients with matching **CyclicBarrierName** parameter does not equal the number of parties. **Parties** is another parameter of distsync sampler client element and *must* be the same for all clients!

The third parameter of distsync sampler client is **RemoteConcurrentFactory** that specifies the location of distsync server (e.g. //**localhost**/**RemoteConcurrentFactory**).

For distsync to work with jMeter you must have a copy of distsync's jar in jMeter's **lib**/**ext**/ directory.

4. Installing and Running the tests

Procedure:

1. Install the **JMeter** application. NOTE: every computer which will function as a test client must have JMeter installed - if you want to run a distributed test, i.e. a test using more than one client, each computer must have a **JMeter** installation in order to to run the jmeter server application. The JMeter server application is required to enable JMeter to communicate with, and remotely start the tests on, the client computer.
2. Set the JMETERROOT environment variable (e.g. **export JMETERROOT=/PATH-TO-JMETER-INSTALLATION**) - where you substitute the actual path on your system for "PATH-TO-JMETER-INSTALLATION")
3. Source the ACS environment (e.g. **. .bash_profile.acs**). NOTE: this step assumes that you have an existing ACS installation; if not, you must install ACS.
4. Retrieve the ACS source from CVS, if you haven't already - e.g. **cvs export ACS**
5. Build (and deploy) the tests, by executing the **ACS**/**Benchmark**/**JMeter**/**build** script. NOTE: this does a **make clean man all install** in the ACS/Benchmark/Components/Performance/src directory, and also does some additional deployment/configuration that is necessary for **JMeter** to run the tests. You must run this script (or manually replicate what it does) on all of the computers which will be clients in your test - e.g. if you want to run a distributed test with multiple clients, you must do this on all of the client computers.
6. Set your ACS_CDB environment variable to point to the test CDB location: (e.g. **export ACS_CDB=PATH-TO**/**ACS**/**Benchmark**/**Components**/**Performance**/**config** where you substitute the actual path on your system for "PATH-TO")
7. Start ACS (e.g. **acsStart**)

8. Start the C++ container named "bilboContainer" (e.g. **acsStartContainer -cpp bilboContainer**)
9. Optionally, you may wish to bring up object explorer to verify that you "see" the test components (e.g. **objexp** followed by browsing the components in the GUI), but this is merely an optional verification step that is not required
10. Configure JMeter's **jmeter.properties** file. Edit this file and modify the **remote_hosts** setting to contain a comma-separated list of the IP addresses of each client computer in your distributed test e.g. "remote_hosts=146.88.1.156,146.88.1.59" where you would use the IP addresses of your client computers in place of the 146.xxx.xx.xx numbers. You may also, optionally, modify the **log_level** setting for Jmeter). NOTE: you only need to modify the jmeter.properties file on the computer which will run the JMeter GUI application to kick off all the tests; you do not need to modify the jmeter.properties file on all of the client computers in your test environment.
11. Run the distsync server on a single computer: **$JMETERROOT/bin/distsync-server 146.88.1.156**, where you substitute the IP address of your machine in place of 146.88.1.156. NOTE: do not use 127.0.0.1 here. Also, you must run this prior to running jmeter-server (see the next step) due to the way the scripts are implemented to share the rmi registry. This script starts the rmi registry, in addition to starting the distsync server. The script will fail if the rmi registry is already executing, which occurs when you run jmeter-server first. Don't do that! The distsync-server is an application which allows the behavior of multiple computers to be synchronized during the tests.
12. Start the JMeter server application: **$JMETERROOT/bin/jmeter-server** on each client computer in your test environment. NOTE: if you are running the tests only on a single machine from within the JMeter GUI application, this step is not strictly necessary. However, for a distributed test, it is mandatory to run this on each of the client computers in the test environment in order for JMeter to properly communicate with each of the test clients.
13. Start the JMeter GUI application by running the **$JMETERROOT/bin/jmeter** script.
14. Load the test plan file by choosing "File->Open" and browsing to the file **ACS/Benchmark/JMeter/acsperftest.jmx** and selecting it in the file chooser dialog.
15. Modify test parameters, as necessary, and optionally disable/enable specific parts of the test plan. You will need to set MANAGER_REFERENCE to point to the computer on which you have ACS running. You will also need to set DISTSYNC_SERVER to point to the computer on which you ran the distsync-server script. You will need to set REMOTE_COUNT to the number of clients in your test. You may also, optionally, modify the THREADS_COUNT, TESTS_SIZE, and LOOP_COUNT variables.
16. Optionally, add/modify listeners and possibly select output file. See JMeter documentation for more details on this process.
17. Start the tests from JMeter GUI application (e.g. from the menu, choose **Run->Remote Start All**).
18. Observe the status of the running tests by expanding the tree on the left in the GUI; ACS Stress Tests->All Tests, then selecting the "View Results in Table" node. You should see

output from the running tests indicating success or any error messages.

Additional precautions and notes:

- Be sure to have **MANAGER**, **DISTSYNC_SERVER** and **REMOTE_COUNT** parameters of jMeter test plan adjusted properly.
- When changing number of involved clients (machines times threads) you *must* also restart **distsync-server**.
- **distsync-server** should also be restarted when test plan is not completed entirely.
- When restarting **distsync-server** and/or **jmeter-server** you may in some cases have to kill rmiregistry process(es) (by **killall rmiregistry** or **stop-rmiregistry** script).

5. Expectations

5.1. Scalability

Scalability is more the question of asynchronous operations, because there we are dealing with so called *open servicing system* - requests may be coming faster than being serviced. Until we are servicing fast enough, average response time of one operation should remain more or less constant, thereafter, *average* response time will instantly raise to infinity => *SYSTEM UNSTABLE*!

Synchronous operations, on the other hand, are *closed/interactive servicing system* that is much more stable. The amount of requests is dependant on the number of clients. If all the clients are waiting to be serviced, no new request can be generated. For as long as waiting queue is empty (we are servicing requests/clients faster as they *return* - in average), we will have constant response times. After that limit is broken, response time will increase lineary with the number of clients.

Response time is the time from the moment we send the request to the moment we receive the response. It, of course, depends on the *queue size/fullness* and the *servicing time* (time for servicing the request).

With our 2.2.* tests we calculate average servicing time. It might also be interesting to find the average response time (by remembering the time of making each request (e.g. by sending it as request id) and subtracting it from the time of response in the callback object and then finding the average of these time differences).

5.2. Effects of different parameters

Size of sets:

- In case of very fast operations we want to perform more (a set) of them together to get more accurate results.
- Until the *constantly intensive* system's situation is not stabilized we are getting shorter response times for a single operation than the average response time actually is

(because at the beginning the request queue is empty). The larger sets we take the more stabilized situation we measure. To get the actual average response times we should take infinitely large sets.

The biggest deviation of measured time from the actual average response time will happen in case of an unstable system - *an overwhelmed open servicing system*. In that case we will always measure a finite response time, although it should be infinite. This conclusion might help us find whether a given system is unstable just by measuring averages for more different-sized sets. If `K` is the size of a set, function `avg_time(K)` should with `K->inf` approach toward some constant in case of a stable system, and should grow to infinity in case of an unstable system.

Number of clients (threads times machines):

- Increases the intensity of requests coming to server. Increase of intensity will have bad impact on response time. Too high intensity will, in case of asynchronous operations, make server become unstable.

Delay between two sequent requests for each client:

- Decreases the intensity of requests coming to server.

6. Results

Please, see **acstests.sxc**!

7. Conclusions

Please, see **acstests.sxc** and **AcsPerformance.pdf**!

7.1. Possible extensions of the tests

Let me just repeat some, that were already mentioned in previous chapters:

- Completing unperformed tests (()).
- Measuring response time of each operation (Scalability (5.1.)).
- Performing scalability tests also with lower incoming request intensities by using delay parameter of asynchronous operations tests (**T22Sampler** in jMeter clients (3.4.)). If milliseconds are too much, **BCTClient** and **T22Sampler** should be modified to take nanoseconds instead.

8. How to extend tests

If you want to add another measurement to the test plan, here is a note on what needs to be done...

From here I will presume you already have Java code that performs the operation (or a set of operation if one takes too little time to perform). First you must create a jMeter class that extends **org.apache.jmeter.protocol.java.sampler.AbstractJavaSamplerClient**. Then this class should be inserted and adjusted properly to fit into the jMeter's plan.

8.1. Creating jMeter sampler client class

Implementing **org.apache.jmeter.protocol.java.sampler.AbstractJavaSamplerClient** class means to implement the following methods:

- `public void setupTest(JavaSamplerContext context);`

  Should do any initialization required by the client. It is generally recommended to do any initialization such as getting *fixed* parameter values here rather than in the **runTest** method in order to add as little overhead as possible to the test. Parameters that vary during the execution of the test plan should, however, be handled in **runTest** method because **setupTest** is only called the first time jMeter plan encounters this test. Test parameters are contained in **context** parameter of this method - see Apache JMeter API Specification [1] on **JavaSamplerContext**.

- `public SampleResult runTest(JavaSamplerContext context);`

  jMeter executes this method every time it needs to perform the test. It expects some attributes of returning **SampleResults** to be defined in here (e.g. **setTime**, **setSuccessful**, **setSampleLabel** and possibly others - see Apache JMeter API Specification [1] on **SampleResult** for more). This method should perform a single sample (operation or a set of operations) for each iteration in jMeter's test plan and measure the time required for it to perform.

- `public void teardownTest(JavaSamplerContext context);`

  Do any clean-up required by this test here. This method is called for all initialized sampler client objects involved in the testing after the whole test plan completes.

- `public Arguments getDefaultParameters();`

  Provide a list of parameters which this test supports. Any parameter names and associated values returned by this method will appear in the GUI by default so the user does not have to remember the exact names. The user can add other parameters which are not listed here. If this method returns `null` then no parameters will be listed. If the value for some parameter is `null` then that parameter will be listed in the GUI with an

empty value.

To keep consistency with other, already created sampler client classes you should start from the following template:

```
package com.cosylab.acs.test.sampler;

import java.util.Iterator;

import org.apache.jmeter.config.Arguments;
import org.apache.jmeter.samplers.SampleResult;
import
org.apache.jmeter.protocol.java.sampler.AbstractJavaSamplerClient;
import org.apache.jmeter.protocol.java.sampler.JavaSamplerContext;

/**
 *
 * DESCRIPTION OF YOUR CLASS!
 *
 */
public class TMySampler extends AbstractJavaSamplerClient
{
	private static final String TEST_PREFIX = "TEST_ID";

	private String m_testName, m_instanceID;

	/**
	 * Do any initialization required by this client. It is
generally
	 * recommended to do any initialization such as getting
parameter
	 * values in the setupTest method rather than the runTest
method
	 * in order to add as little overhead as possible to the test.
	 *
	 * @param context  the context to run with. This provides
access
	 *                  to initialization parameters.
	 */
	public void setupTest(JavaSamplerContext context)
	{
		// NOTE: This function is only called once for one
```

```
thread
                getLogger().debug(whoAmI() + "\tsetupTest()");
                listParameters(context);

                long tn = context.getLongParameter("ThreadNum", 0);

                m_testName = TEST_PREFIX;
                try {
                        String str =
java.net.InetAddress.getLocalHost().getHostName();
                        m_instanceID = str + ":";
                } catch (Exception e) {}
                m_instanceID += tn;
        }

        /**
         * Perform a single sample.
         * Perform a single sample for each iteration.  This method
         * returns a <code>SampleResult</code> object.
         * <code>SampleResult</code> has many fields which can be
         * used.  At a minimum, the test should use
         * <code>SampleResult.setTime</code> to set the time that
         * the test required to execute.  It is also a good idea to
         * set the sampleLabel and the successful flag.
         *
         * @param context  the context to run with. This provides
access
         *                 to initialization parameters.
         *
         * @return         a SampleResult giving the results of this
         *                 sample.
         */
        public SampleResult runTest(JavaSamplerContext context)
        {
                SampleResult results = new SampleResult();

                try
                {
                        long time = System.currentTimeMillis();

                        // PERFORM TESTING OPERATION(S) HERE!
```

```
                        results.setTime(System.currentTimeMillis() -
time);
                        results.setSuccessful(true);
                        results.setSampleLabel(m_testName + "() @" +
m_instanceID);

                        if (orb != null)
                                orb.destroy();
                }
                catch (Exception e)
                {
                        results.setSuccessful(false);
                        results.setResponseCode(e.getMessage());
                        results.setSampleLabel("ERROR: " +
e.getMessage());
                        getLogger().error(this.getClass().getName() +
": Error during sample", e);
                }

                if (getLogger().isDebugEnabled())
                {
                        getLogger().debug(whoAmI() + "\trunTest()" + "\
tTime:\t" + results.getTime());
                        listParameters(context);
                }

                return results;
        }

        /**
         * Do any clean-up required by this test.
         *
         * @param context  the context to run with. This provides
access
         *                  to initialization parameters.
         */
        public void teardownTest(JavaSamplerContext context)
        {
                // NOTE: This function is only called once for one
thread
                getLogger().debug(whoAmI() + "\tteardownTest()");
                listParameters(context);
        }
```

```
	/**
	 * Provide a list of parameters which this test supports.  Any
	 * parameter names and associated values returned by this
method
	 * will appear in the GUI by default so the user doesn't have
	 * to remember the exact names.  The user can add other
parameters
	 * which are not listed here.  If this method returns null
then
	 * no parameters will be listed.  If the value for some
parameter
	 * is null then that parameter will be listed in the GUI with
	 * an empty value.
	 *
	 * @return  a specification of the parameters used by this
	 *           test which should be listed in the GUI, or null
	 *           if no parameters should be listed.
	 */
	public Arguments getDefaultParameters()
	{
		Arguments params = new Arguments();
		params.addArgument("ThreadNum", "${__threadNum}");
		return params;
	}

	/**
	 * Dump a list of the parameters in this context to the debug
log.
	 *
	 * @param context  the context which contains the
initialization
	 *                 parameters.
	 */
	private void listParameters(JavaSamplerContext context)
	{
		if (getLogger().isDebugEnabled())
		{
			Iterator argsIt =
context.getParameterNamesIterator();
			while (argsIt.hasNext())
			{
				String name = (String)argsIt.next();
```

```
                                getLogger().debug(name + "=" +
context.getParameter(name));
                        }
                }
        }

                                                /**
         * Generate a String identifier of this test for debugging
         * purposes.
         *
         * @return  a String identifier for this test instance
         */
        private String whoAmI()
        {
                StringBuffer sb = new StringBuffer();
                sb.append(Thread.currentThread().toString());
                sb.append("@");
                sb.append(Integer.toHexString(hashCode()));
                return sb.toString();
        }

}
```

Text in **bold** should be replaced by your code. The *emphasised* code could just as well be left out because it is there for jMeter logging purposes only. In order to make use of this logging, you should adjust jMeter's **jmeter.properties** file to have **log_level.jmeter.protocol.java.sampler** set to **DEBUG** mode and everything will get logged into file defined by **log_file.jmeter** variable.

You may also add your own test parameters by (optionally) defining them in **getDefaultParameters()** method and using them the same way we use **ThreadNum** parameter.

See Apache JMeter API Specification [1] for more details on how you may retrieve different types of parameters from **JavaSamplerContext**.

If you merge more different tests into one java sampler client class, you should have each identify itself through **m_testName** variable as in **T21Sampler.java**.

Compiled classes (jar files) then need to be placed in jMeter's **lib/ext/** directory.

8.2. Creating jMeter test plan

You may add your test to existent **acstests.jmx** document or you can create a new plan. Opening one **.jmx** document in jMeter does not close what is already open but rather only appends the plan to the open one.

When adding to **acstests.jmx** try to keep the consistncy with the rest of the plan. One of the important things is also having all the tests take similar amount of time to execute. These

execution times should be correalated with **TESTS_SIZE** parameter of the test plan wherever possible.

For tests that will be performed by more threads or even machines concurrently you should first have them synchronized with one distsync sampler client object.

That being said, lets now see how these things are put in jMeter:

- To append a new element as a child of selected one either right-click the selected one and choose Add or select Edit - Add from the menu.
- Samplers (elements that perform one specific test - e.g. implemented as jMeter sampler client class) can only be appended to **Thread Group** or one of the **Logic Controller** elements. Named elements control the flow of the test plan (looping, ...).
- To add a jMeter sampler client object as a sampler you should do Add - Sampler - Java Request and then select the Classname (of the added element) to match your test's classname (your test's jar file must of course be located in jMeter's **/lib/ext** directory - jMeter's classpath). Be sure to adjust your test's parameters.
- To add distsync sampler, you should do Add - Sampler - Java Request and then as Classpath select **com.cosylab.distsync.JMeterSampler**. **Parties** parameter must match the number of clients (i.e. number of computers times number of threads per computer). Do not forget to adjust **RemoteConcurrentFactory** parameter to match location of the DistSync server (more about this has been said in DistSync (3.6.)).
- For more on how to use jMeter, please, refer to Apache JMeter User's Manual [2] .

Document History

Revision	Date	Author	Section	Modification
1.0	2004-03-12	Anze Zagar	all	Created.
	2004-03-21	Klemen Zagar	all	
1.1	2004-08-13	Steve Harrington	all	Clarified and updated to reflect inclusion in ACS build environment.

References

ID	Author	Reference	Revision	Date	Publisher
1	Apache	Apache JMeter API Specification	1.9	2003	Apache
2	Apache	Apache JMeter User's Manual		2003	Apache
3	Apache	Apache JMeter	1.9	2003	Apache