Francesc Alted

OpenLC User’s Guide

Alted, Francesc:

OpenLC User's Guide

Day of print: June, 24th

Contents

I've made this longer than usual because I
lack the time to make it shorter.

—Blaise Pascal

Chapter 1

Introduction

1.1 Description

OpenLC is a set of software tools designed to facilitate benchmarking and stress testing of a wide variety of information servers (such as web, email, FTP, LDAP, databases, and so on). The package is built around a microkernel that contains basic routines for benchmarking tasks, such as accessing intermediate results in real-time (*spying* the run data), setting up simulated clients, defining scenarios, handling database calls, comparing results of different runs, summarizing data, etc.

OpenLC also offers an API for developers interested in creating clients (called *commanders*, to distinguish them from the simulated client inside the microkernel) that query services provided within the microkernel.

1.2 Architecture

OpenLC is designed around an open client-server architecture. It is open in that communication between the server and the different clients conforms to the XML-RPC protocol (http://www.xml-rpc.org). In principle, OpenLC implements XML whenever data is interchanged among server and client. This means that any language that supports XML-RPC can be used to develop a *commander* (see later about the meaning of the *commander* word in OpenLC context).

Nevertheless, in the case of internal servers, modules written in Python (possibly with C extensions if performance is a must) are preferred, due mainly to the convenience of using the same language to access to the microkernel services.

The diagram 1.1 shows the OpenLC architecture. In the next sections we will describe the different components.

1.2.1 Commander

The commander is a client that allows the end user to define run scenarios and to execute commands that control the run. It may be written to use any communication protocol supported by the different external servers. A developer may choose to use the current XML-RPC external server, and design commanders that use the already-defined API (see `doc/microkernel-API.txt`) to talk with the microkernel via this external server.

The scenario definition file (**SDF**)is made using an XML configuration file (see client/config-http.xml for an example), but the commander could offer a graphical interface to facilitate the data introduction.

Included with OpenLC there's a test commander (**OLCCommander**) that connects to the server and performs several tasks, such as sending a run scenario, starting and stopping a run, sampling real-time data, requesting run statistics or plotting the final results. Read the source code of this commander to see a sample implementation of the API.

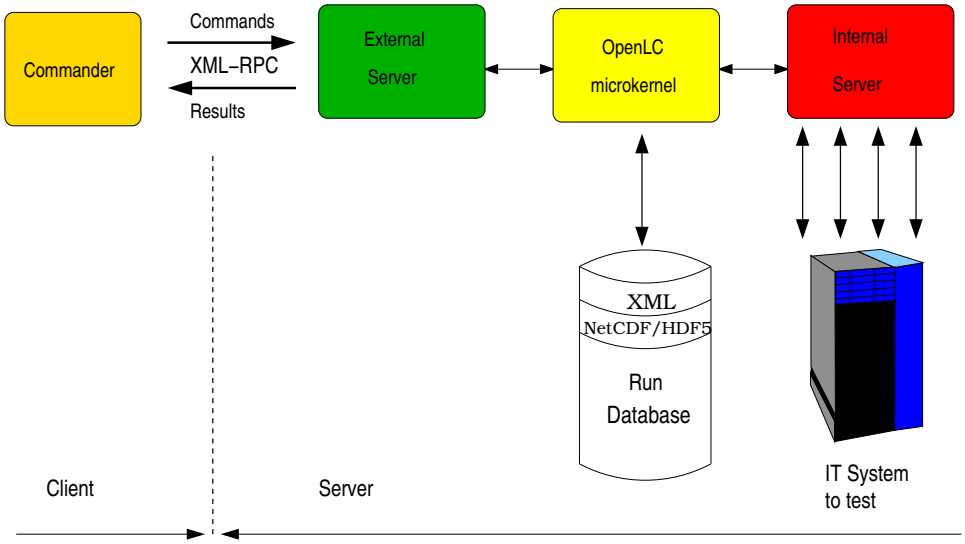

Figure 1.1: Diagram showing the OpenLC architecture

Anyone interested in contributing to the OpenLC project can start by writing new commanders that add functionality and use the API in new and interesting ways. Right now, the commanders can be written in any language (provided they have an XML-RPC library and they conform to the API).

Now, let's see the different OpenLC Server components more in depth. This is a completely optional section. You can skip that and go directly to the Installation section in chapter 2.

1.2.2 External server

The XML-RPC external server (*server/controlCore.py* and *server/baseCore.py*) is the primary point of contact for every client. It exposes a XML-RPC API to clients, and coordinates all traffic between clients and the OpenLC microkernel. In short, the external server is the interface that connects the microkernel to the world (via the microkernel API).

Currently there is only one external server exposing the microkernel API using the XML-RPC communication protocol. There may be (and will be) more than one external server. However, the XML-RPC server will remain (hopefully for long time) a central piece to access to microkernel functionality. In the future, I may add a SOAP external server.

1.2.3 Microkernel

The microkernel (*server/Core.py*) is the heart of OpenLC: it is the microkernel that contains all the necessary logic for easily creating extensible benchmarking tools.

The microkernel is in charge of parsing the scenario (sent by the commander), setting up the simulated service clients (currently I'm using threads, but that may change in the future), starting and stopping them, then collecting the runtime data and sending it to the external server and database management module. It coordinates all the necessary actions to serve requests coming from commanders. In addition, it will be in charge of logging all the requests and actions done to easy a debugging session.

1.2.4 Database management module

The database management module (*server/manageData.py*) provides three simple methods to handle data gathered by the service clients: **putData** and **getData** to respectively save and retrieve data objects. Currently the data repository is built on top of a mixture of XML (for python objects), NetCDF and HDF5 (for numeric

data) database. This combination forms a very powerful object-oriented database with a high degree of transparency (from the programmer point of view), robustness, and efficency.

1.2.5 Internal server

Finally, the internal server interprets the commands in the run scenario and does the **real** work by generating load on the target systems. There will be (one or several, depending on the goals of the desired load) different internal servers for each type of service it generates the load. For example, there is already an internal server for HTTP and FTP protocols and another for IMAP4.The purpose of an internal server is mainly for encapsulate functionality and to protect the microkernel for having too much complexity. This will allow also the making of new internal servers without interfere with the existing ones.

In this version, three internal servers are provided:

- *Local:* It **only** simulates the response times of an hypothetic server by using a determined function that return synthetic response times. So, there is no need to load an exterior IT server, and it's perfect to do local tests and check if everything in the OpenLC machinery is sane and well. This mode is also good during the commander developing process where synthetic times are enough (sometimes preferred) to test client functionality. More information in section 4.2.1.
- *HTTP:* Simulate real HTTP 1.0 clients (and shortly 1.1), by mapping a thread for each simulated client. Each thread request the different URL as stated on the scenario input file, and returns the response times. If Python is configured with OpenSSL support, secure HTTP (https://name) is also supported. In addition, FTP, Gopher and file:// protocols are supported by this internal server. See section 4.2.2for details.
- *IMAP4:* Simulate IMAP4rev1 clients, by mapping a thread for each simulated client. At the moment, no all IMAP4 commands are supported. Please, see section 4.2.3.

All the internal servers reads his configuration and task list from the scenario provided by the commander (through the microkernel).

In the coming releases I'll be adding more internal servers, which will provide support to protocols like SMTP, POP3, LDAP, and so on.

1.3 Final words (or *Call for hackers*)

Despite of its alpha status, we think OpenLC is quite easy to use and to extend. The goal of the OpenLC project is to develop a lightweight, powerful and stable load-testing microkernel with a flexible API (see `doc/microkernel-API.txt`) that can be easily adapted to a wide variety of environments, users, services and needs.

If you have any suggestion, bug report or anything related with OpenLC, I'll be happy to hear about you!.

All conviction -- serious conviction -- will become a prejudice to posterior convictions. In the end, there's no need for too much convictions for life. There's enough with three or four. Not more.

—Joan Fuster

Chapter 2

Installation

2.1 Prerequisites

OpenLC is programmed in Python. The main development platform is Linux, but OpenLC and the software it depends upon also should run on other Unices and Windows platforms. Provided your machine is properly configured with the necessary base software, there's no reason why it shouldn't run on your platform. If it doesn't, please send me a mail.

In the next sections, very detailed prerequisite installation instructions are described. Please, read and follow carefully the guidelines stated there before to continue.

2.1.1 Prerequisites for running OpenLC for UNIX flavor

Here you will find brief instructions for installing the required software in UNIX platforms. First are listed the requisites for OpenLC server, and then for the client.

To compile and install everything, you will need the GNU C compiler, GNU Make, autoconf, automake and other common utilities. You will find that there are also binaries packages ready for Linux or Solaris, but installing them in that form is not as fun as compiling the sources :-).

Although the development platform for OpenLC is Linux, we have been able to compile (and execute) successfully the prerequisites and OpenLC in Solaris 7 (UltraSparc). Other Unices should also compile and execute well, but your mileage may vary. Please, drop me an e-mail if you get into difficulties.

Please, unless you know what you are doing, install the packages preferably in sequential order. If you have installation problems with a specific software, refer to the original package documentation. Note also that, in general, version numbers are stated because they run well with OpenLC. You may use other versions and probably you can have success for running OpenLC, but in some cases don't. If you are experiencing problems, please, double-check the prerequisite versions, and try to stay in sync with them.

Mandatory prerequisites for the OpenLC server

1. *Python* 2.1 or higher: It works with both Python 2.1.x and Python 2.2.x (highly recommended 2.1.3 or 2.2.1). Normally you can find binaries for your platform in `http://www.python.org`.

2. *Python XMLRPC* 0.8.8.2 or higher: Get and unpack sources from `http://sourceforge.net/projects/py-xmlrpc`, then:

```
cd py-xmlrpc-0.8.8.2
python setup.py build
(become root)
python setup.py install
```

3. *Numeric Python (NumPy)* 21.0 or higher: Get and unpack sources from `http://sourceforge.net/projects/numpy`, then:

```
cd Numeric-21.0
python setup.py build
(become root)
python setup.py install
```

4. *NetCDF* 3.5.0 library: Get and unpack sources from `http://www.unidata.ucar.edu/packages/netcdf`, then:

```
./configure
make
make test
(become root)
make install
```

5. *Scientific Python* 2.3.6 or higher: Get and unpack sources from `ftp://dirac.cnrs-orleans.fr/pub/ScientificPython/`, then:

```
cd ScientificPython-2.3.6
python setup.py build
(become root)
python setup.py install
```

6. *Gnosis utils* 1.0.2 or higher: Get and unpack sources from `http://www.gnosis.cx/download/`, then:

```
cd Gnosis_Utils-1.0.2
python setup_gnosis.py build
(become root)
python setup_gnosis.py install
```

Optional prerequisites for the OpenLC server

HDF5 is a general purpose library and file format for storing scientific data, and some of the most powerful data analysis packages (for example, **R** and **octave**) can read it directly. As the HDF5 FAQ says:

> HDF5 can store two primary objects: datasets and groups. A dataset is essentially a multidimensional array of data elements, and a group is a structure for organizing objects in an HDF5 file. Using these two basic objects, one can create and store almost any kind of scientific data structure, such as images, arrays of vectors, and structured and unstructured grids. You can also mix and match them in HDF5 files according to your needs.

If you want to have the run data (raw and reduced) in HDF5 format in addition to NetCDF, you have to install a couple of packages more.

1. *HDF5* 1.4.3 or higher: Get and unpack sources from `ftp://hdf.ncsa.uiuc.edu/HDF5/`, then:

```
cd hdf5-1.4.3
./configure
make
(become root)
make install
# Make this library accessible. In linux this is normally
```

```
# acomplished by issuing an ldconfig command, but this is
# OS dependent.
# Check documentation about shared libraries for
# more information, such as the ld(1) and ld.so(8)
# manual pages.
ldconfig   # If Linux
```

2. *HL-HDF* 0.40 or higher: Get and unpack sources from `ftp://ftp.ncsa.uiuc.edu/HDF/HDF5/contrib/hl-hdf5/README.html`, then:

```
cd hlhdf-0.40
./configure
make
(become root)
# The setup.py is in the debian directory, but is completely general
# for any UNIX or Windows OS.
python debian/setup.py install
```

Mandatory prerequisites for the OpenLC client

For running the client **OLCCommander** you will need *Python* 2.1 or higher, as well as the *Python XMLRPC* 0.8.8.2 or higher. Follow the installation instructions as in the server case.

2.1.2 Prerequisites for running OpenLC for Windows flavor

Although the development platform for OpenLC is Linux, OpenLC, both server and client, also run in Windows without modification because it's a pure Python application. However, there aren't Windows binary versions for all the prerequisites, so I will discuss only how to install prerequisites only for the client program. If you want to install the server, you must have a C compiler (preferably Visual C++ 6.0), GNU Make, autoconf and other common utilities. You may have a look at the excellent package MinGW (Minimalist GNU for Windows) for free and GNU compatible tools. Read the section 2.1.1 for detailed compilation and installation instructions.

Have in mind that, unless you know what you are doing, you should install the packages preferably in sequential order. If you have installation problems with a specific software, refer to the original package documentation.

Prerequisites for the OpenLC client

1. *Python* 2.1 or higher: I highly recommended 2.1.3 or 2.2.1. You can find binaries for Windows in `http://www.python.org`.

2. *Python XMLRPC* 0.8.8.2 or higher: Fetch and install binaries from `http://sourceforge.net/projects/py-xmlrpc`.

3. *Gnosis utils* 1.0.2 or higher: There isn't a binary version. Get and unpack sources from `http://www.gnosis.cx/download/`, then:

```
cd Gnosis_Utils-1.0.2
python setup_gnosis.py build
(become super-user, if needed)
python setup_gnosis.py install
```

2.2 Installing OpenLC

OpenLC is pure Python code, with no C extensions, so no C compiler is needed. To install it, get the package from `http://openlc.sf.net`, unpack the sources, and proceed as follows:

```
cd OpenLC-0.6
# Edit OpenLC-config.xml and choose the rundata directory.
# Edit setup.py and select the directories for configuration
# files and executables.
python setup.py build
(become super-user)
python setup.py install
```

Once you have done that, you are ready to run OpenLC server (**OLCServer** command) and client (**OLCCommander** command). Of course, you can choose to run both on the same or different machines. See the next chapter to see some examples of use.

... durch planmässiges Tattonieren.
... through systematic, palpable experimentation.
[asked how he came upon his theorems]

—Karl Friedrich Gauss

Chapter 3

Getting Started

3.1 A first example

To start with, go to the the server machine and start the OpenLC server. To do this, simply run this command:

```
$ OLCServer
Initialization done. Ready to serve requests.
On UNIX, type 'q' and 'Return' to exit. On Windows, kill this python process.
```

The server initializes the run database then enters a loop and waits for IP connections. By default it listens in the 43434 port (you can change it by using the `-p <port>` option). You can halt the server by pressing 'q' and <return>. On Windows your only chance to stop OpenLC server is by killing that python process using the process management tool (hot-key: `ctrl-alt-supr`).

Once the server is up and running, run (in a different window or machine) the sample commander (**OLCCommander**) together with a scenario to do some tests. This client reads the scenario file passed in the command line, and some calls to the microkernel API are made using XML-RPC. You can start a test run, by issuin the next command line:

```
$ OLCCommander -s 1 sdf/local-config.xml
```

This is a completely automatic test so no interaction is needed from the user except optional modification of scenario file. Now, look at the output of this example and the Scenario Definition File (SDF) used; then read some explanations on what has happened.

```
Scenario file:  local-config.xml
Preparing the server:
['OK', 'Mon Jun 17 18:06:28 2002']
sending start:
['OK', 'Mon Jun 17 18:06:28 2002']
{   'Local/test/constant': {   'commandNumber': 1,
                               'dataTransferred': 10.082274999999999,
                               'threadNumber': 3,
                               'timeSpent': 0.0098460000000000006,
                               'wallClock': 0.99921199999999999},
    'Local/test/linear': {   'commandNumber': 3,
                             'dataTransferred': 10.573729999999999,
                             'threadNumber': 3,
                             'timeSpent': 0.010326,
                             'wallClock': 0.98901300000000003}}
```

```
{   'Local/test/constant': {   'commandNumber': 1,
                               'dataTransferred': 9.8293459999999993,
                               'threadNumber': 3,
                               'timeSpent': 0.0095989999999999999,
                               'wallClock': 1.9887189999999999},
    'Local/test/linear': {   'commandNumber': 3,
                             'dataTransferred': 10.418213,
                             'threadNumber': 3,
                             'timeSpent': 0.010174000000000001,
                             'wallClock': 1.9787710000000001}}
{   'Local/test/constant': {   'commandNumber': 1,
                               'dataTransferred': 9.8078610000000008,
                               'threadNumber': 3,
                               'timeSpent': 0.0095779999999999997,
                               'wallClock': 2.9786049999999999},
    'Local/test/linear': {   'commandNumber': 3,
                             'dataTransferred': 10.838013,
                             'threadNumber': 3,
                             'timeSpent': 0.010584,
                             'wallClock': 2.999555}}
{   'Local/test/constant': {   'commandNumber': 1,
                               'dataTransferred': 10.085326999999999,
                               'threadNumber': 3,
                               'timeSpent': 0.0098490000000000001,
                               'wallClock': 3.9992260000000002},
    'Local/test/linear': {   'commandNumber': 3,
                             'dataTransferred': 10.590210000000001,
                             'threadNumber': 3,
                             'timeSpent': 0.010342,
                             'wallClock': 3.989023}}
{   'Local/test/constant': {   'commandNumber': 1,
                               'dataTransferred': 10.493895999999999,
                               'threadNumber': 3,
                               'timeSpent': 0.010248,
                               'wallClock': 4.9981999999999998},
    'Local/test/linear': {   'commandNumber': 3,
                             'dataTransferred': 9.5662839999999996,
                             'threadNumber': 3,
                             'timeSpent': 0.0093419999999999996,
                             'wallClock': 4.987603}}
sending Stop:
['OK', 'Mon Jun 17 18:06:33 2002']
sending Finish:
['OK', 85]
sending getStatistics:
[OK]
-- Some statistical results coming from post-processing run data --
-- Grand total for the entire run (runID --> 85) --
Elapsed Time:       5.01 s
Cummulated Elapsed Time:      24.01 s
Number of Transactions:       2475
Response Time (mean):    0.00970 s
Transaction Rate:     494.06 trans/s
```

```
Total Data Transferred   24586.67 (KB)
```

Let's have a look at the scenario definition file (sdf/local-config.xml):

```
<configuration>
  <clients number="5"></clients>
  <time max="5"></time>
  <sample small="yes" period="2.0"></sample>
<!-- The number of bins of the resulting reduced data -->
  <reduction minElementsPerBin="10" maxBins="100"></reduction>
<!-- Here begins the protocol-specific definition -->
  <runProtocol name="Local" mode="random">
<!-- Maximum number of iterations for this protocol and for each client -->
<!-- All the cmd's hanging from here are considered one single iteration -->
    <iterations max="1000"></iterations>
<!-- values we are interested in -->
    <retValues>
      <wallClock units="s" sta="mean" typecode="Float"></wallClock>
      <timeSpent units="s" spyplot="yes" sta="vds" typecode="Float"></timeSpent>
      <dataTransferred units="KB" sta="vds" typecode="Float"></dataTransferred>
      <commandNumber units=" " typecode="Int"></commandNumber>
      <threadNumber units=" " sta="minimal" typecode="Int"></threadNumber>
    </retValues>
    <cmdGroup name="test">
      <cmd spy="yes" name="constant">
        constant(range=0.01)
      </cmd>
      <cmd spy="no" name="random">
        random(range=0.01)
      </cmd>
      <cmd spy="yes" name="linear">
        linear(range=0.01)
      </cmd>
    </cmdGroup>
  </runProtocol>
</configuration>
```

3.2 Quick explanation

Before a command is issued into the server, a message is printed telling what the commander is about to do. For example, the first thing it does is to send a *getReady* **RPC** to the server (indicated by a `Preparing the server` message). The next line prints the results from this RPC call. Right now, this is a 2 elements tuple, telling you if the call has been ok ("OK") or not ok ("NO"), and a timestamp of **when** the command has been executed in the *server*.

After *getReady*, a *startRun* command is sent ("`sending start`"). We can see that it was executed without problems in the server.

Then, as we called the `OLCCommander` with the `-s 1` option which means that we wanted to *spy* the data every 1 seconds, the client sent a *getSampleData* RPC command each 1 seconds. Please, don't confuse this time period with the `period` attribute of the `<sample>` scenario tag. This will be used in the server in future releases, but not now. For every *getSampleData* command issued, a hash is printed with information about the very last commands executed on the server. In this example, we collect information about the `Local/test/constant` and `Local/test/linear` commands, because we asked for this info in the

scenario definition (see the *spy* attribute for these commands). For each measurement collected, it will be displayed all its attributes, which are selected on the `<retValues>` tag subtree (see chapter 4).

If you choose *sample.small="no"*, you would get the pure raw data for each measurement, which would be quite a lot of information. The small attribute is introduced to easy the management of big quantities of data for spying purposes on the client side. If you select the *"no"* value, be sure to redirect the standard output of `OLCCommander` to a file so as to not mess too much your shell terminal.

Next, a *stopRun* command is sent, followed by an *finishRun*. The return value for the *finishRun* call tell us the run number assigned to this run (instead of a normal timestamp). Note it down for future references to this run because right now no functionality to retrieve past run data from the client is implemented, so you will have to go to the server and look at the data on the run database.

Then, a *getStats* is sent to the server to collect some statistics of this last run. They are printed nicely by the printStatistics() method. This information is normally what the user want to have a quick idea of what happened on the stress test and to extract preliminary conclusions. If you want more information you can always go to the rundata repository (on the server side) and look carefully at the **raw data** and **reduced data** files (see chapter 5).

It is worth noting that this *sample commander* is a **real** commander. It basically uses all the present microkernel API, but, of course, its functionality is limited. If you browse the `OLCCommander` source you will find how easy is to talk with the OpenLC server.

Feel free to modify the parameters in the scenario definition file and experiment with different values, then save the file and rerun the client to see the changes. Try also with the http-config.xml scenario sample. Read chapter 4 for more information about the different parameters and its values.

Things should be as simple as possible,
but not simpler.

—Albert Einstein

Chapter 4

Scenario Definition File (SDF) Format

The scenario definition file is an XML file which determines all the parameters which are needed in the experiment. The XML format was chosen because of its multiples advantages:

- The XML structure is hierarchical, which is very apropiate to define subscenarios.
- When a DTD is provided, a commander can force the use of the different parameters (i.e. you don't have to send the file to the server in order to see if it's DTD conformant or not).
- The use of a DTD will easy the use of XML editors in order to guide the user in defining the scenario parameters.
- It's a standard, and there are XML parsers for a lot of languages. So developers can use a wide language range in order to build a commander.

The OpenLC scenario definition files are evolving rapidly, getting richer in features and parameters. This is because they have to adapt constantly to the new features coming up in the OpenLC server. So, don't expect to have always the same tags in the scenario. However, their definition will significantly freeze as OpenLC approaches his stable release (1.0), while I'll try hard to keep these changes under a minimum. No DTD's are defined at this moment (most probably will be in the next release).

Right now, three different XML scenario files are implemented (and are coming in future releases). One is used for define a *Local* experiment, other for a *HTTP* load experiment and the third for a *IMAP4* experiment. Before to state the differences, we will start with a brief explanation to the general parameters which are common to all the scenarios.

4.1 General scenario tags

Let's return again to the XML file used in the section 3.1. We will comment it by signalling the main properties for each of the XML tags in the file[1]. When the tags are not general it will be noted.

Let's start from the beginning.

configuration

```
<configuration>
  <clients number="5"></clients>
  <time max="5"></time>
  <sample small="yes" period="2.0"></sample>
<!-- The number of bins of the resulting reduced data -->
  <reduction minElementsPerBin="10" maxBins="100"></reduction>
```

[1] Be careful with the letter cases in the next XML examples, because both XML and Python are case SENSITIVE.

```
.
.
.
</configuration>
```

As you see, the file always start by a **<configuration>** tag and is closed by his counterpart </configuration>. Those tags are mandatory and signals the start and the end of the document (this is called the *root* element).

Let's see the tags which are direct descendents of **configuration** tag:

- The **<clients>** tag fixes the number of clients in this experiment (more attributes may come in the future). It only accepts one attribute, *"number"*, and in this documentation we usually refer to it as *clients.number* variable. This notation is convenient because it is used internally in the OpenLC code, and also reflects the hierarchical nature of the scenario structure. In this case, we are telling the OpenLC server that we want a total of 5 simulated clients (currently, they are implemented as Python threads).

- *time.max* variable: the maximum execution time (wall clock time) for all the run expressed in seconds. The value is 5 seconds in this case. The run will finish wherever a *time.max, runProtocol.iterations.max* is reached, or when a finishRun command is received by the server, the event which happens first.

- *sample.small* variable: it is used to select the amount of data to be sampled on the spy process. If true ("yes" value), a call to *getSampleData* only returns the latest transaction gathered by the server. If false ("no" value), all the data gathered from the previous call to the getSampleData() is returned.

- **<reduction>** tag: it defines values for the data reduction process which is carried out on the server side. This process consists basically in computing histograms for each of the variables measured for each command, group and protocol along the wallclock time axis. The *reduction.maxBins* variable sets the maximum number of bins for the histograms, and the *reduction.minElementsPerBin* the minimum number of data measurements included in each bin.

runProtocol

```
<runProtocol name="Local" mode="random">
  <iterations max="1000"></iterations>
```

This tag marks the start of a *protocol* definition. That is, the data values we want to save, the maximum iterations, groups of commands and, of course, the proper commands we want to issue in order to refine the scenario in the protocol context, so we can properly call this a *subscenario*. Some subscenarios are very simple ones (i.e., *Local*), but, in general, they may be very rich in subtags and attributes depending on the internal servers to deal with.

Now, we will explain the main characteristics for some **runProtocol** components:

- *name*: Attribute selecting which protocol (i.e. which internal server) we want to invoke. For the moment, three protocols are supported: *Local*, *HTTP* and *IMAP4*.

- *runProtocol.mode*: This attribute can have two values:

  - *"normal"* (default): all the commands behind (**<cmd>** tags) will be executed sequentially in each thread.

  - *"random"*: randomizes the sequence of commands.

- *iterations.max*: sets the maximum number of iterations for this protocol. All the cmd's hanging from **runProtocol** are considered one single iteration.

retValues

```
<retValues>
  <wallClock units="s" sta="mean" typecode="Float"></wallClock>
  <timeSpent units="s" spyplot="yes" sta="vds" typecode="Float"></timeSpent>
  <dataTransferred units="KB" sta="vds" typecode="Float"></dataTransferred>
  <commandNumber units=" " typecode="Int"></commandNumber>
  <threadNumber units=" " sta="minimal" typecode="Int"></threadNumber>
</retValues>
```

This tag selects the variables we want to save for each transaction. In this example, we can choose between the next variables (but this may depend on the internal server implementation used for the protocol):

- *wallClock*(mandatory): selects (for saving)the time (from the beginning of the run) when the transaction has been completed.
- *timeSpent*: selects the time that the transaction has taken.
- *dataTransferred*: gives the amount of data transferred during the transaction.
- *commandNumber*: is the number of command in the command's list.
- *threadNumber*: indicates which thread number was responsible of the transaction.

In the *retValue*'s subtags, we see that we have some attibutes to specify more clearly its properties.

- *units* sets the unit of measurement for the variable (s for seconds, KB for kilobytes and so on).
- *typecode* sets the number type (right now only 'Int' or 'Float').
- *sta* which sets the type of statistics we want to extract from data. They can be:
  - *vds*: Very Detailed Statistics
  - *mean*: Only mean values
  - *minimal*: No histogram, only a mean value
  - *none*: No histogram, no mean value

cmdGroup

```
<cmdGroup name="test">
```

The **<cmdGroup>** tag groups several command (**<cmd>**) tags to form "atomic" actions to be done. The *runProtocol.mode* variable described above doesn't interfere inside this groups, where the execution is guaranteed to be sequential. This feature is perfect to simulate a Web shopping procedure, IMAP4 procedure call sequence or, in general, a group of actions you want to ensure they will always be done sequentially. Also, the microkernel do *command group* statistics based on this tag grouping.

It has only one attribute for the moment:

- *cmdGroup.name* variable: The name of the group for future references on the run database and the statistics.

cmd

```
    <cmd spy="yes" name="constant">
      constant(range=0.01)
    </cmd>
    <cmd spy="no" name="random">
      random(range=0.01)
    </cmd>
    <cmd spy="yes" name="linear">
      linear(range=0.01)
    </cmd>
  </cmdGroup>
 </runProtocol>
</configuration>
```

Finally, here we have the most internal tag for **<runProtocol>**, the **<cmd>** tag. Here goes all the stuff related with the actual commands issued by the internal servers to attack (or just **simulate** this, in the case of *Local* protocol) exterior IT servers. It has several components:

- *name* attribute: The name of the command for future references on the run database and the statistics.
- *spy* attribute: A boolean variable indicating if we want to *spy* this command or not. Values: *"yes"* / *"no"*. Default value: *"no"*.
- *PCDATA* (only one line allowed): The PCDATA (i.e. the text between the `<xmltag atr1=" ", atr2=" ", ...>` and `</xmltag>` tags), selects the procedure (chosen from a range in the internal server API) to attack the exterior server. In this case, three different procedures will be invoked: *constant*, *random* and *linear*. For an explanation on what this procedures actually do, see the *Local protocol* section. As you can see, you can pass parameters to the internal servers procedure.

In the next section, we will have a more-in-depth look into the different sub-scenarios currently implemented in the OpenLC server.

4.2 Subscenario definitions

In the last section we have made a description on the general tags for OpenLC scenarios. But one of the OpenLC's strengths is the flexibility to adapt to an array of stress testing situations. In this section we will discuss the different XML subscenarios for the internal servers present in the OpenLC server.

4.2.1 Local subscenario

The Local scenario is implemented as a stand-alone and synthetic test. It is very useful to simulate runs and the user is offered the capability to control a variety of parameters to test the OpenLC capabilities (for example, how many transactions per second can deal with), to quickly design and test new clients, or just to learn using it.

In this subscenario, the only components which are different from the general format are the *Local* server internal API. Right now there are three procedures:

- **constant(range=floatValue)**: Sleep for a number of seconds given by the *range* parameter.
- **linear(range=floatValue)**: Sleep for a number of seconds between 0 and the value of the *range* paramenter. This number increases linearly during the duration of the run.
- **random(range=floatValue)**: Sleep for a random number of seconds, in the range of [0..*range*].

All this procedures also return a *dataTransferred* **retValue** which is simply the time spent by the sleep call multiplied by a factor (right now 1024).

4.2.2 HTTP (FTP, Gopher, file) subscenario

Like the *Local* protocol, no special tags are needed, and the only procedure implemented right now on the HTTP internal server API is *get*.

- **get(url="stringValue")**: Get the URL stated in the url variable. It is based on the urllib Python module so, it supports HTTP 1.0 and secure HTTP (https://name) if Python is configured with OpenSSL. It also supports the FTP, Gopher and file protocols.

4.2.3 IMAP4 subscenario

The support for the IMAP4 protocol adds a couple of tags in order to authenticate the users before doing any transaction. The tags are:

- **auth**: This tag allows the input of authorization info for the host and users. It has one attribute, *host* which takes the value of the machine to send requests to.
- **user**: It is a subtag of **auth** and it's intended to provide the username (*user.name* variable) and password (*user.password* variable) information for each user.

The IMAP4 internal server connects to *auth.host* and authentifies each thread with an *user.name* identifier. The map between the threads and usernames is made using a round-robin algorithm. If there are more clients than user names, several clients use the same username (some IMAP4 server support until 4 sessions with the same user). If there are more usernames that clients, there will remain usernames unused.

The IMAP4 internal server API is basically the same supported by the *imaplib* Python module (see `http://www.python.org/doc/lib/module-imaplib.html`. Right now, an effort is made to compute the *dataTransferred* (**retValue**) value, but take this as an aproximation until a better algorithm is implemented.

In any field find the strangest thing and then explore it.

—John Archibald Wheeler

Chapter 5

Diving Into The OpenLC Run Data

Generally speaking you don't need to further post-process the data obtained in an OpenLC run. There are times, however, in which you would like to analize carefully which are the response times in iteration #N, or simply, access to data which is not printed by default after a `OLCCommander` execution.

5.1 OpenLC Run Database Outline

OpenLC saves, by default, all the run (both *raw* an *reduced*) data on a directory in the server side. In this chapter I will try to tell you where and how access to this internal data.

As you probably already know (you have configured it), the rundata is saved in the directory stated in the `OpenLC-config.xml` (normally in `/etc/OpenLC` directory), under the tag **rundbpath**. The structure of this directory is outlined in figure 5.1.

As you can see, the OpenLC run database is a two-level hierarchical directory. In the first level, there exist a directory for each run number (in the form `(run%06d % runNumber)`). In the second level (i.e. for each run), a number of files are created containing all the data for the current run.

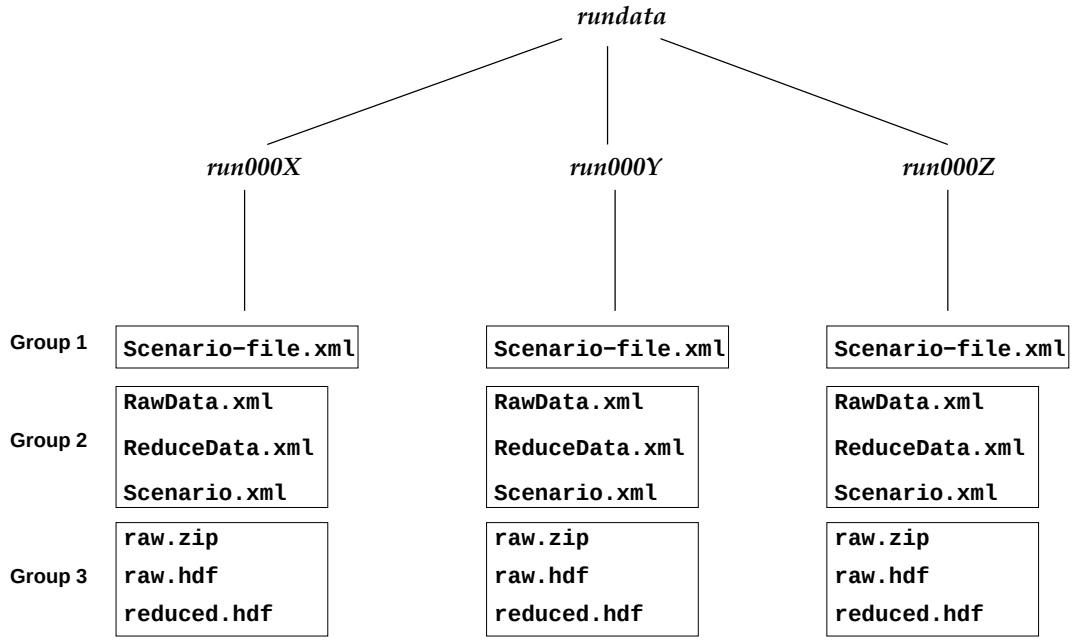

Figure 5.1: Diagram showing the OpenLC database structure

These second-level files are grouped in three subsets:

- *Group 1*: Only one file is included here, the `Scenario-file.xml`, and as you may guess, it's the Scenario Definition File responsible for this run.
- *Group 2*: This group includes important Python objects in XML serialization form (using the Gnosis utilities). These objects are important to retrieve important information of the run (for example, the number of transactions or the actual run duration time). As the format in which the objects are saved is XML, you can read the values by just look at the file, which can be an important advantage.
  Three different files are present in this group:
  - `RawData.xml`
  - `ReduceData.xml`
  - `Scenario.xml`

  Look at their contents to have an idea on what you can find there.
- *Group 3*: The data coming along from the different transactions during the run are saved in NetCDF and HDF5 (if configured during installation) formats. As we stated before this detailed information may be very useful to dive into the details of the run.
  Three different files are present in this group:
  - `raw.zip`: This is a ZIP file which contains the raw data (in NetCDF format) for each command separately. This files are really the original file streams where the data is saved in real-time (using the Scientific Python package) during the run.
  - `raw.{hdf|nc}`: This is the same data than above, but consolidated in a unique file. If HDF support is there, the `raw.hdf` contains the arrays of command data structured in a tree. If HDF is not supported, a `raw.nc` is generated, with the same information, but without a tree structure and the variable names are constructed from the protocol, group and command names (for example `Local-test-random-wallClock`). This format is normally more difficult to manage than the tree form.
  - `reduced.{hdf|nc}`: This file holds the reduced data resulting from a post-processing raw data process which takes place when the run is finished. The format of this data depends on the values stated in the **retValues** tag of the SDF file. If no HDF support, a NetCDF file is created with this same information (but with flat structure).

5.2 Browsing at the Run *Raw Data*

If you want to have look at the data collected by the experiment without further processing, you can start with `RawData.xml`. Here you have an example of the information you can find there:

```
<!DOCTYPE PyObject SYSTEM "PyObjects.dtd" [<PyObject class="RawData" module="OpenLC.server
<attr type="string" name="hdfFilename" value="/home/falted/OpenLC/rundata/run000097/raw.hd
<attr type="numeric" name="runid" value="97"></attr>
<attr type="numeric" name="nevents_saved" value="2505"></attr>
<attr type="numeric" name="nevents" value="2505"></attr>
<attr type="numeric" name="elapsedTime" value="5.0204830169677734"></attr>
<attr type="dict" name="filenames" id="136977668">
  <entry>
    <key type="string" value="Local/test/random"></key>
    <val type="string" value="/home/falted/OpenLC/rundata/run000097/raw/Local/test/random.
  </entry>
  <entry>
```

```
    <key type="string" value="Local/test/constant"></key>
    <val type="string" value="/home/falted/OpenLC/rundata/run000097/raw/Local/test/constant
  </entry>
  <entry>
    <key type="string" value="Local/test/linear"></key>
    <val type="string" value="/home/falted/OpenLC/rundata/run000097/raw/Local/test/linear.nc
  </entry>
</attr>
<attr type="dict" name="lastindex" id="138290436">
  <entry>
    <key type="string" value="Local/test/random"></key>
    <val type="numeric" value="835"></val>
  </entry>
  <entry>
    <key type="string" value="Local/test/constant"></key>
    <val type="numeric" value="835"></val>
  </entry>
  <entry>
    <key type="string" value="Local/test/linear"></key>
    <val type="numeric" value="835"></val>
  </entry>
</attr>
<attr type="string" name="zipfile" value="/home/falted/OpenLC/rundata/run000097/raw.zip"></a
<attr type="string" name="raw_dir" value="/home/falted/OpenLC/rundata/run000097/raw"></attr>
<attr type="list" name="cmdIDs" id="137022540">
  <item type="string" value="Local/test/constant"></item>
  <item type="string" value="Local/test/random"></item>
  <item type="string" value="Local/test/linear"></item>
</attr>
<attr type="string" name="currentDir" value="/home/falted/OpenLC/rundata/run000097"></attr>
<attr type="string" name="datetime" value="Sun Jun 23 12:13:56 2002"></attr>
</PyObject>
```

As you can see, you can get some interesting information here. If you look carefully at **filenames** tag, you will discover that it's a mapping (*dictionary* in Python jargon) between the commands in the Scenario and the NetCDF filenames. The files referenced by these filenames are package inside the `raw.zip` file.

In raw.hdf you can get all the raw data structured in a tree, which is well adpated to show the Scenario description for this run. The `.hdf` extension means that the file is in HDF5 format and there is a variety of software and utilities to read it. I would recommend you a couple: *HDF5* tools (`http://hdf.ncsa.uiuc.edu/hdf5tools.html`) and *HDFView* (`http://hdf.ncsa.uiuc.edu/hdf-java-html/hdfview/`). If you don't have HDF5 support, I suggest you to have a look at the utilities *ncdump* and *ncgen* which comes with the NetCDF library or the excellent plotting tool called *grace* (`http://plasma-gate.weizmann.ac.il/Grace/`).

Next, I'll describe shortly some of these utilities.

5.2.1 HDF5 tools

There is quite a few utilities to deal with HDF5 files in *HDF5 tools*, but the most important ones are `h5dump` and `h5ls`. You can get instructions by passing them the flag `-?`. As an example of use, look at the next command and its output:

```
$ h5ls -r raw.hdf
/raw.hdf/Local           Group
/raw.hdf/Local/test      Group
```

```
/raw.hdf/Local/test/constant Group
/raw.hdf/Local/test/constant/commandNumber Dataset {835}
/raw.hdf/Local/test/constant/dataTransferred Dataset {835}
/raw.hdf/Local/test/constant/threadNumber Dataset {835}
/raw.hdf/Local/test/constant/timeSpent Dataset {835}
/raw.hdf/Local/test/constant/wallClock Dataset {835}
/raw.hdf/Local/test/linear Group
/raw.hdf/Local/test/linear/commandNumber Dataset {835}
/raw.hdf/Local/test/linear/dataTransferred Dataset {835}
/raw.hdf/Local/test/linear/threadNumber Dataset {835}
/raw.hdf/Local/test/linear/timeSpent Dataset {835}
/raw.hdf/Local/test/linear/wallClock Dataset {835}
/raw.hdf/Local/test/random Group
/raw.hdf/Local/test/random/commandNumber Dataset {835}
/raw.hdf/Local/test/random/dataTransferred Dataset {835}
/raw.hdf/Local/test/random/threadNumber Dataset {835}
/raw.hdf/Local/test/random/timeSpent Dataset {835}
/raw.hdf/Local/test/random/wallClock Dataset {835}
/raw.hdf/info            Group
```

5.2.2 HDFView

A nicer utility is HDFView. This is a Java application which lets you browse the contents of the HDF5 file, and even visualize dataset histograms. From the HDFView manual:

> The HDFView is a Java-based tool for browsing and editing NCSA HDF4 and HDF5 files. HDFView allows users to browse through any HDF4 and HDF5 file; starting with a tree view of all top-level objects in an HDF file's hierarchy. HDFView allows a user to descend through the hierarchy and navigate among the file's data objects. The content of a data object is loaded only when the object is selected, providing interactive and efficient access to HDF4 and HDF5 files. HDFView editing features allow a user to create, delete, and modify the value of HDF objects and attributes.

In figure 5.2 you can see an example of an HDFView session.

5.2.3 ncdump

This utility is very useful to browse NetCDF files. From the manual:

> The ncdump tool generates the CDL text representation of a netCDF file on standard output, optionally excluding some or all of the variable data in the output. The output from ncdump is intended to be acceptable as input to ncgen. Thus ncdump and ncgen can be used as inverses to transform data representation between binary and text representations. ncdump may also be used as a simple browser for netCDF data files, to display the dimension names and sizes; variable names, types, and shapes; attribute names and values; and optionally, the values of data for all variables or selected variables in a netCDF file.

5.3 The *Reduced Data* at a Glance

When the run is finished, the OpenLC server automatically starts a process to reduce the *raw data* and extract some run data statistics (following instructions in the scenario file) and saves it in files. You can find two files with the post-processed data information.

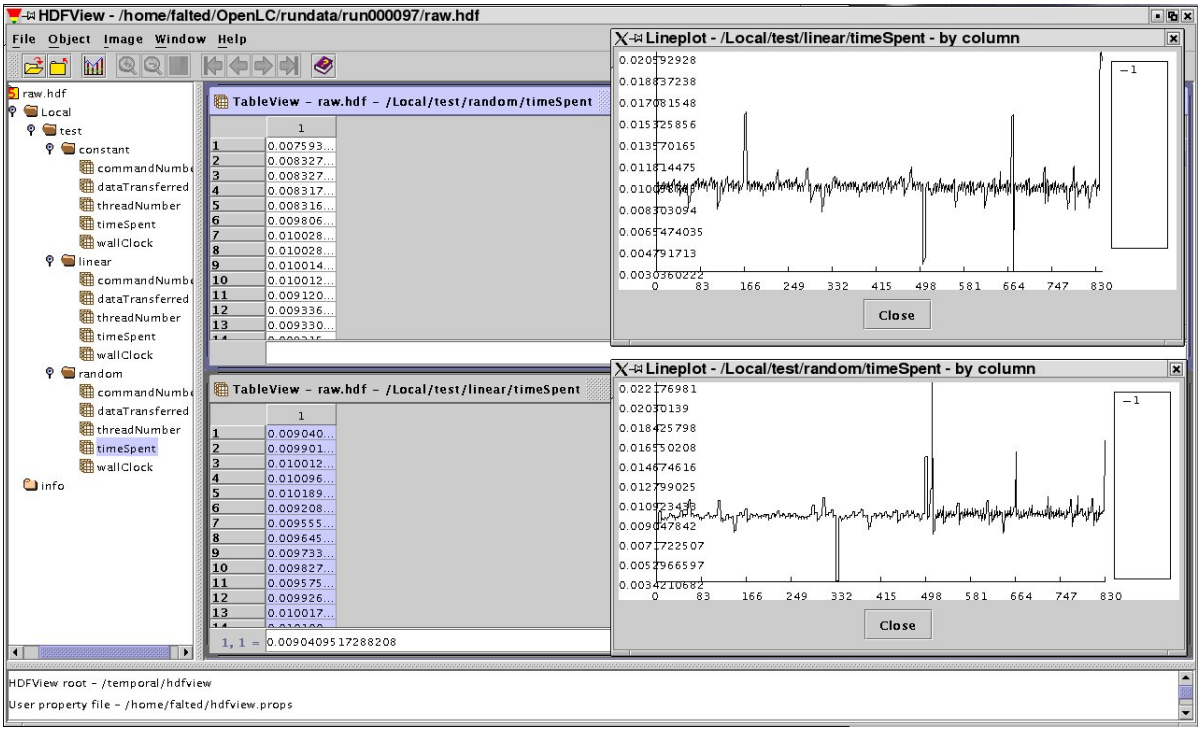

Figure 5.2: An HDFView session screenshot.

- `ReduceData.xml`: This file is the serialization of the ReduceData internal server class. It has a serialized hash holding mean values for all the variables for each command, group and protocol defined in the Scenario.

- `reduced.hdf`: Contains, depending on the *sta* attribute of the **retValues** subtags, the next datasets for the commands, groups and protocols:

  – *vds*: Stands for *Very Detailed Statistics* and computes histograms for mean, nevents, maximum, minimum and standard deviation for bins along the *wallClock* variable.

  – *mean*: Computes histogram only for *mean* values.

  – *minimal*: Computes noly the mean in all the *wallClock* range. The result is an scalar, not a histogram.

  – *none*: Ignore this variable in the final reduced data output.

  The format of this file is HDF5 (or NetCDF if HDF5 is not supported), so to dive in you can use the same tools than reported in section 5.1.