Distributed Computing with Xenon

tutorial

Jurriaan H. Spaaks Jason Maassen

November 24, 2015

Contents

1 Introduction

Many scientific applications require far more computation or data storage than can be handled on a regular PC or laptop. For such applications, access to remote storage and compute facilities is essential. Unfortunately, there is not a single standardized way to access these facilities. There are many competing protocols and tools in use by the various scientific data centers and commercial online providers.

As a result, application developers are forced to either select a few protocols they wish to support, thereby limiting which remote resources their application can use, or implement support for all of them, leading to excessive development time.

Xenon is a library designed to solve this problem. It offers a simple, unified programming interface to many remote computation and data storage facilities, and hides the complicated protocol and tool specific details from the application. By using Xenon, the application developer can focus on the application itself, without having to worry about which protocols to support, resulting in faster application development.

1.1 Purpose of this document

This document aims to help users without much prior knowledge about Java programming and without much experience in using remote systems to understand how to use the Xenon library.

1.2 Version information

It is assumed that you are using one of the Ubuntu-based operating systems (I'm using Linux Lubuntu 14.10). Nonetheless, most of the material covered in this manual should be usable on other Linux distributions with minor changes. The manual is written to be consistent with Xenon release 1.1.1[1].

[1]For releases, see `https://github.com/NLeSC/Xenon/releases`

2 Conceptual overview

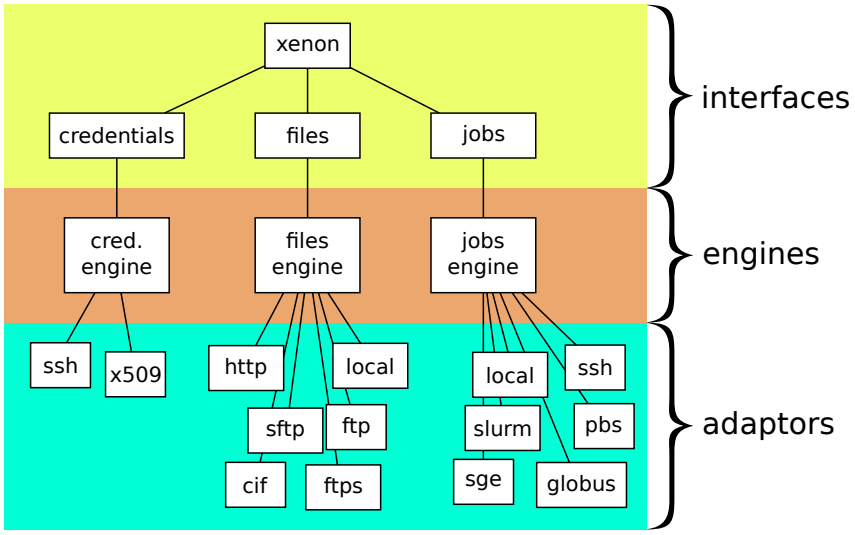

Figure 1: Xenon is built on 3 pillars: 'credentials', 'files' and 'jobs'. Each pillar consists of an interface, an engine, and an adaptor.

3 Basic usage

To use a minimal feature set of the Xenon library, you'll need the following software packages:

1. *Git*, a version management system;
2. *Java*, a general purpose programming language;

The following sections describe the necessary steps in more detail.

3.1 Installing Git

Open a terminal (default keybinding Ctrl + Alt + t). The shell should be Bash. You can check this with:

```
echo $0
```

which should return `/bin/bash`.

Now install `git` if you don't have it already:

```
sudo apt-get install git
```

After the install completes, we need to get a copy of the Xenon software. We will use `git` to do so. Change into the directory that you want to end up containing the top-level repository directory. I want to put the Xenon stuff in my home directory, so for me that means:

```
cd ${HOME}
```

Then clone the Xenon repository into the current directory:

```
git clone https://github.com/NLeSC/Xenon.git
```

This will create a directory `~/Xenon` that contains the Xenon source code.

3.2 Installing Java

Xenon is a Java library, therefore it needs Java in order to run. Java comes in different versions identified by a name and a number. The labeling is somewhat confusing[1]. This is partly because Java was first developed by Sun Microsystems (which was later bought by Oracle Corporation), while an open-source implementation is also available (it comes standard with many Linuxes). Furthermore, there are different flavors for each version, each flavor having different capabilities. For example, if you just want to *run* Java applications, you need the JRE (Java Runtime Environment); if you also want to *develop* Java software, you'll need either an SDK (Software Development Kit) from Sun/Oracle, or a JDK (Java Development Kit) if you are using the open-source variant.

Check if you have Java and if so, what version you have:

```
java -version
```

That should produce something like:

```
java version "1.7.0_79"
OpenJDK Runtime Environment (IcedTea 2.5.6) (7u79-2.5.6-0ubuntu1.14.04.1)
OpenJDK 64-Bit Server VM (build 24.79-b02, mixed mode)
```

[1] See for example `http://stackoverflow.com/questions/2411288/java-versioning-and-terminology-1-6-vs-6-0-openjdk-vs-sun`

Note that 'Java version 1.7' is often referred to as 'Java 7'.

If you don't have Java yet, install it with:

```
sudo apt-get install default-jdk
```

3.3 Verifying the software setup

To check if everything works, we first need to build the example from source and then run the example from the command line.

3.3.1 Building Xenon

The Xenon repository includes a file `gradlew` in the root directory. `gradlew` is a useful little program, and we will discuss it in more detail later. For now, we will just use `gradlew` to compile the Java source code we need to run the examples, as follows:

```
cd ${HOME}/Xenon
./gradlew examplesClasses
```

This should give you two new directories ~/`build/classes/examples/` and ~/`build/classes/main/` with compiled Java classes in them.

As a test, we will use Xenon to list the contents of the current directory. The magic incantation to do so consists of 4 parts:

1. the word `java` invokes the Java program;
2. the classpath option `-cp` followed by a colon-separated list of paths, defining a list of locations where `java` is allowed to look for Java classes;
3. the Java class we want to run. The class name needs to be fully qualified and should be present in one of the directories listed in the classpath;
4. the input arguments to the Java class, in our case `file://$PWD`.

So, putting all that together you get:

```
cd ${HOME}/Xenon
java -cp 'build/classes/examples:build/classes/main:lib/*' \
nl.esciencecenter.xenon.examples.files.DirectoryListing file://$PWD
```

which should list the contents of the current directory when you run it.

3.3.2 Simple Xenon program from the command line

Now for the actual example. The sourceSet 'main' contains the source code for Xenon. It is located at `src/main/java`. We'll also need a second sourceSet, 'examples', which contains the source code for the Xenon examples. We'll need to compile both sourceSets in order to run a simple Xenon Java program.

Let's first check what tasks we have by:

```
cd ${HOME}/Xenon
./gradlew tasks --all
```

Under 'Build tasks', there should be an item `examplesClasses`, used for compiling the 'examples' sourceSet; `examplesClasses` has a dependent task `classes`, used for compiling the 'main' sourceSet.

Running

```
cd ${HOME}/Xenon
./gradlew examplesClasses
```

should give you a new directory `~/Xenon/build` with subdirectories `classes/examples` and `classes/main` (as well as some other stuff).

The general syntax for running compiled Java programs from the command line is as follows:

```
java <fully qualified classname>
```

The fully qualified classname for our example is `nl.esciencecenter.xenon.examples.CreatingXenon`, but if you try to run

```
cd ${HOME}/Xenon
java nl.esciencecenter.xenon.examples.CreatingXenon
```

you will get the error below:

```
Error: Could not find or load main class \
nl.esciencecenter.xenon.examples.CreatingXenon
```

This is because the `java` executable tries to locate our class `nl.esciencecenter.xenon.examples.CreatingXenon`, but we haven't told `java` where to look for it. We can resolve that by specifying a list of one or more directories using `java`'s classpath option `-cp`. There are 3 locations that are relevant for running `CreatingXenon`. These are:

1. the location of `CreatingXenon` itself:
   `~/Xenon/build/classes/examples`

2. the location of the Xenon classes:
   `~/Xenon/build/classes/main`

3. the location of any libraries that Xenon depends on:
   ~/Xenon/lib

These directories can be passed to `java` as a colon-separated list. Directory names can be relative to the current directory. Furthermore, the syntax is slightly different depending on what type of file you want `java` to find in a given directory: if you want `java` to find compiled Java classes, use the directory name; if you want `java` to find jar files, use the directory name followed by `/*`. Finally, the order within the classpath is significant.

Using paths relative to `~/Xenon` for items (1) and (2) above, and using the `/*` addition for item (3) yields the following classpath value for our example: `build/classes/examples:build/classes/main:lib/*`, so the whole command becomes:

```
cd ${HOME}/Xenon
java -cp build/classes/examples:build/classes/main:lib/* \
nl.esciencecenter.xenon.examples.CreatingXenon
```

Your output should look something like this:

```
13:21:15.594 [Thread-0] DEBUG n.e.xenon.engine.util.CopyEngine - CopyEngin ...
13:21:15.606 [main] DEBUG n.e.xenon.engine.util.JobQueues - Creating JobQu ...
13:21:15.618 [main] DEBUG n.e.xenon.adaptors.ssh.SshAdaptor - Setting ssh  ...
13:21:15.632 [main] DEBUG n.e.xenon.adaptors.ssh.SshAdaptor - Host keys in ...
13:21:15.643 [main] DEBUG n.e.xenon.adaptors.ssh.SshAdaptor - |1|x5PcOam9h ...
13:21:15.650 [main] DEBUG n.e.xenon.adaptors.ssh.SshAdaptor -
13:21:15.650 [main] DEBUG n.e.xenon.adaptors.ssh.SshAdaptor - Setting ssh  ...
13:21:15.657 [main] WARN  n.e.xenon.adaptors.ssh.SshAdaptor - OpenSSH conf ...
java.io.FileNotFoundException: /home/daisycutter/.ssh/config (No such file or directory)
    at java.io.FileInputStream.open(Native Method) ~[na:1.7.0_79]
    at java.io.FileInputStream.<init>(FileInputStream.java:146) ~[na:1.7.0_79]
    at java.io.FileInputStream.<init>(FileInputStream.java:101) ~[na:1.7.0_79]
    at com.jcraft.jsch.Util.fromFile(Util.java:492) ~[jsch-0.1.50.jar:na]
    at com.jcraft.jsch.OpenSSHConfig.parseFile(OpenSSHConfig.java:97) ~[jsch-0.1.50.jar:na]
    at nl.esciencecenter.xenon.adaptors.ssh.SshAdaptor.setConfigFile(SshAdaptor.java:192) [main/:na]
    at nl.esciencecenter.xenon.adaptors.ssh.SshAdaptor.<init>(SshAdaptor.java:164) [main/:na]
    at nl.esciencecenter.xenon.adaptors.ssh.SshAdaptor.<init>(SshAdaptor.java:141) [main/:na]
    at nl.esciencecenter.xenon.engine.XenonEngine.loadAdaptors(XenonEngine.java:182) [main/:na]
    at nl.esciencecenter.xenon.engine.XenonEngine.<init>(XenonEngine.java:169) [main/:na]
    at nl.esciencecenter.xenon.engine.XenonEngine.newXenon(XenonEngine.java:92) [main/:na]
    at nl.esciencecenter.xenon.XenonFactory.newXenon(XenonFactory.java:57) [main/:na]
    at nl.esciencecenter.xenon.examples.CreatingXenon.main(CreatingXenon.java:39) [examples/:na]
Exception in thread "main" java.lang.NoClassDefFoundError: org/globus/tools/proxy/GridProxyModel
    at nl.esciencecenter.xenon.adaptors.gftp.GftpAdaptor.<clinit>(GftpAdaptor.java:47)
    at nl.esciencecenter.xenon.engine.XenonEngine.loadAdaptors(XenonEngine.java:187)
    at nl.esciencecenter.xenon.engine.XenonEngine.<init>(XenonEngine.java:169)
    at nl.esciencecenter.xenon.engine.XenonEngine.newXenon(XenonEngine.java:92)
    at nl.esciencecenter.xenon.XenonFactory.newXenon(XenonFactory.java:57)
    at nl.esciencecenter.xenon.examples.CreatingXenon.main(CreatingXenon.java:39)
Caused by: java.lang.ClassNotFoundException: org.globus.tools.proxy.GridProxyModel
    at java.net.URLClassLoader$1.run(URLClassLoader.java:366)
    at java.net.URLClassLoader$1.run(URLClassLoader.java:355)
    at java.security.AccessController.doPrivileged(Native Method)
    at java.net.URLClassLoader.findClass(URLClassLoader.java:354)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:425)
```

```
    at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:308)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:358)
    ... 6 more
```

3.3.3 Checking the connectivity

Before we can run the `CreatingXenon` example, we first have to make sure that we have access to a remote system. You'll need an account on the remote machine. For example, I have an account `jspaaks` on SURFsara's Lisa clustercomputer. Cluster computers typically have a dedicated machine (the so-called 'headnode') that serves as the main entry point when connecting from outside the cluster. For Lisa, the headnode is located at `lisa.surfsara.nl`.

I can connect to Lisa's head node using the `ssh` command line program as follows:

```
# (my account on Lisa is called jspaaks)
ssh jspaaks@lisa.surfsara.nl
```

If this is the first time you connect to the remote machine, it will generally ask if you want to add the remote machine to the list of 'known hosts'. For example, here's what the Lisa system tells me when I try to ssh to it:

```
The authenticity of host 'lisa.surfsara.nl (145.100.29.210)' can't be
established.
RSA key fingerprint is b0:69:85:a5:21:d6:43:40:bc:6c:da:e3:a2:cc:b5:8b.
Are you sure you want to continue connecting (yes/no)?
```

If I then type `yes`, it says[1]:

```
Warning: Permanently added 'lisa.surfsara.nl,145.100.29.210' (RSA) to
         the list of known hosts.

                                <some content omitted>
```

and asks for my password.

The result of this connection is that you should now have a (hidden) directory `.ssh` in your `/home` directory, which should contain 3 files: `id_rsa`, which contains your private RSA key(s); `id_rsa.pub`, which contains your public RSA key(s); and `known_hosts`, which contains a list of systems that you have successfully connected to in the past. `known_hosts` uses one line per known

[1] SURFsara publish RSA key fingerprints for their systems at `https://userinfo.surfsara.nl/systems/shared/ssh`. The number posted there should be the same as what you have in your terminal.

system, and each line begins with the following elements:

- `1` a flag signifying that the third element (host name) is hashed using the SHA1 algorithm;
- `x5PcOam9hhAjdF84++EKwodUNgQ` the (public) salt used to encrypt the host name;
- `NK1rAZev7rV6JSTIdM3ymPpKlQ0` the (hashed) host name;
- key-value pairs, e.g. the RSA fingerprint of the Lisa system `ssh-rsa b0:69:85:a5:21:d6:43:40:bc:6c:da:e3:a2:cc:b5:8b`.

Xenon uses `known_hosts` to automatically connect to a (known) remote system, without having to ask for credentials every time.

Normally, you'd build Xenon while connected to the Internet. The build tool we use is called Gradle. When Gradle then downloads whatever additional software it needs. Gradle will first try to download such packages from MavenCentral[1] (a website that hosts many common Java packages, in many different versions); if the package is not available from MavenCentral, or if the download fails for some other reason, Gradle tries a different website (Bintray[2]). The `repositories` section in `build-common.gradle` lists the repositories that Gradle will try to connect to.

It is also possible to build Xenon while disconnected from the Internet, but in order for that to work, you need to have run `./gradlew` at least once before (while connected to the Internet). This ensures that the necessary Gradle plugins, as well as any libraries that Xenon is dependant on, will have been downloaded.

In order to facilitate both online and offline building, we chose to divide the Gradle work over three files, located in the root of the repository:

1. `build.gradle`
2. `build-offline.gradle`

[1] `https://repo1.maven.org/maven2`
[2] `https://bintray.com/bintray/jcenter`

3. `build-common.gradle`

`build.gradle` and `build-offline.gradle` can be called directly as argument to `./gradlew` (or `gradle`, for that matter); `build-common.gradle` is not intended to be called directly (it should only get called from within either `build.gradle` or `build-offline.gradle`, through the use of `apply from` lines. Deferring to `build-common.gradle` avoids duplication of any tasks that are the same, regardless of whether the build is offline or online.

In this section, we will test the software setup by running a small example, `CreatingXenon`. `CreatingXenon` establishes a connection to a remote system, does something simple, and returns.

Index