Distributed Computing with Xenon

tutorial

Jurriaan H. Spaaks

December 4, 2015

Contents

Chapter 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].

1.3 Conceptual overview

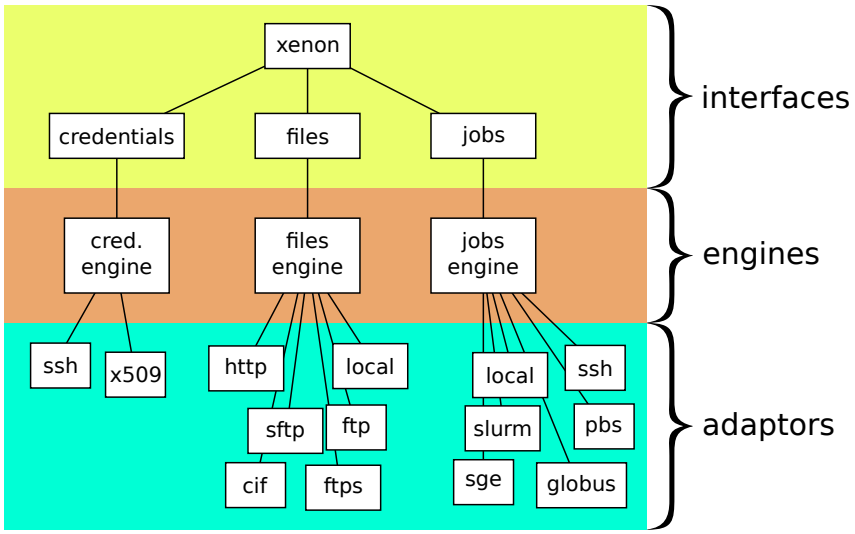

Figure 1.1: Xenon is built on 3 pillars: 'credentials', 'files' and 'jobs'. Each pillar consists of an interface, an engine, and an adaptor.

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

Chapter 2

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.

2.1 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 examples. 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 examples in my home directory, so for me that means:

```
cd ${HOME}
```

Then clone the Xenon-examples repository into the current directory:

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

This will create a directory ~/Xenon-examples that contains the source code of the Xenon examples.

2.2 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)
```

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
```

this will install the open-source variant of Java ('OpenJDK').

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

2.3 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.

Building with `gradlew`

At this point, ~/`Xenon-examples` only contains files directly related to the source code of the example files. However, in order to build and run the examples successfully, we'll need a few more things. Naturally, we'll need a copy of the Xenon library, but the Xenon library in turn also has dependencies that need to be resolved. Because the process of fitting together the right libraries is quite a lot of work, we have automated it. For this, we use the build automation tool Gradle[1]. Interestingly, you do not need to install Gradle for it to work (although you do need Java). This is because the Xenon-examples repository already includes a script called `gradlew`, which will download a predefined version of the Gradle program upon execution. The advantage of using `gradlew` for this is that the resulting build setup will be exactly the same as what the developers use, thus avoiding any bugs that stem solely from build configuration differences.

The `gradlew` script can be run with arguments. For example, running

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

prints a list of the dependencies (of which there are quite many), and

```
./gradlew clean
```

cleans up the files pertaining to any previous builds.

`dependencies` and `clean` are referred to as 'Gradle tasks', 'build tasks' or just 'tasks'. Tasks are defined in a so-called build file called 'build.gradle'. To get an overview of all available tasks you could read through 'build.gradle', or you could simply run:

```
./gradlew tasks
```

or

[1] `http://gradle.org/`

```
./gradlew tasks --all
```

If you run `./gradlew tasks`, you'll see a line at the top that says that the default task is called `shadowJar`[1]. `shadowJar` bundles any and all dependencies into one large jar, sometimes referred to as a 'fat jar'. Once you figure out what to put in it, using a fat jar makes deployment more robust against forgotten-dependency errors. Lucky for you, someone has already figured out how to make the fat jar, and has even written it down in a file that Gradle can understand.

Run `./gradlew` (without any arguments) to start the default task. The following things will happen:

1. the correct version of Gradle will be downloaded;
2. the Xenon library will be downloaded;
3. any dependencies that the Xenon library has will be downloaded;
4. all of that will then be compiled;
5. a fat jar will be created.

Running an example

So at this point we have compiled the necessary Java classes; now we need to figure out how to run them.

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

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

We will use the `DirectoryListing` example from the `nl.esciencecenter.xenon.examples.files` package. As the name implies, it lists the directory contents of a given directory. The fully qualified classname for our example is `nl.esciencecenter.xenon.examples.files.DirectoryListing`, but if you try to run

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

[1] `https://github.com/johnrengelman/shadow`

you will get the error below:

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

This is because the `java` executable tries to locate our class `nl.esciencecenter.xenon.examples.files.DirectoryListing`, 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`.

Directory names can be passed to `java` as a colon-separated list, in which 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 the name of the jar (or use the wildcard `*` if you want `java` to find any jar from a given directory). Finally, the order within the classpath is significant.

We want Java to find the fat jar 'Xenon-examples-all.jar' from 'build/libs'. Using paths relative to `~/Xenon`, our classpath thus becomes `build/libs/Xenon-examples-all.jar`. However, if we now try to run

```
cd ${HOME}/Xenon
java -cp 'build/libs/Xenon-examples-all.jar' \
nl.esciencecenter.xenon.examples.files.DirectoryListing
```

it still does not work yet, because `DirectoryListing` takes exactly one input argument that defines the location (URI) of the directory whose contents we want to list. URIs generally consist of a *scheme* followed by the colon character `:`, followed by a *path*[1]. For a local file, the scheme is `file` (or equivalently, `local`). The path is the name of the directory we want to list the contents of, such as `$PWD` (the present working directory).

Putting all that together, we get:

```
cd ${HOME}/Xenon
java -cp 'build/libs/Xenon-examples-all.jar' \
nl.esciencecenter.xenon.examples.files.DirectoryListing file:${PWD}
```

If all goes well, you should now see the contents of the current directory.

[1]The full specification of a URI is (optional parts in brackets):
`scheme:[//[user:password@]domain[:port]][/]path[?query][fragment]`

Chapter 3

Advanced usage

So now that we've verified that everything works, we can start thinking about doing some development work. Let's first look at opening the Xenon project in the Java editor Eclipse.

3.1 Eclipse

Eclipse is a very powerful, free, open-source, integrated development environment for Java (and many other languages). It is available in most Linuxes from their respective repositories. By default, Eclipse comes with many features, such as Git (version control), Mylyn (task management), Maven (building), Ant (building), an XML editor, as well as some other stuff. While these features are nice, they can get in the way if you're new to code development with Java using Eclipse. We will therefore set up a minimal Eclipse installation which includes only the Eclipse platform and the Java related tools (most importantly, the debugger). Feel free to skip this next part if you're already familiar with Eclipse.

A minimal Eclipse installation

Go to `http://download.eclipse.org/eclipse/downloads/`. Under 'Latest release', click on the link with the highest version number. It will take you to a website that has a menu in the upper left corner. From that menu, select the item 'Platform Runtime binary', then download the file corresponding to

your platform (for me, that is `eclipse-platform-4.5-linux-gtk-x86_64.tar.gz`). Go back to the menu by scrolling up, then select the item 'JDT Runtime binary', and download the file (there should be only one; for me that is `org.eclipse.jdt-4.5.zip`).

Now go to where you downloaded those two files. Uncompress `eclipse-platform-4.5-linux-gtk-x86_64.tar.gz` and move the uncompressed files to a new directory `~/opt/minimal-eclipse/` (they can be anywhere, really, but `~/opt` is the conventional place to install user-space programs on Linux). Start Eclipse by running `eclipse` from `~/opt/minimal-eclipse/eclipse`.

In Eclipse's menu go to Help, then select Install New Software.... Near the bottom of the dialog, uncheck Group items by category. Then click the top-right button labeled Add... and click Archive.... Then navigate to the second file you downloaded, `org.eclipse.jdt-4.5.zip` and select it. In the dialog, a new item Eclipse Java Development Tools should appear. Make sure it's checked, then click Next and Finish. When Eclipse restarts, you should have everything you need for Java development, without any of the clutter!

Adding a Bash alias

Adding a Bash alias to `~/.bash_aliases` will make it easier to start the program. I've used

```
echo "alias miniclipse='${HOME}/opt/minimal-eclipse/eclipse/eclipse'" >> \
${HOME}/.bash_aliases
```

to do so (restart your terminal to use the `miniclipse` alias).

Automatic project setup with `gradlew`

Normally, when you start a new project in Eclipse, it takes you through a series of dialogs to set up the Eclipse project in terms of the directory structure, the classpath, etc. The configuration is saved to (hidden) files `.project`, `.classpath`, and `.settings/org.eclipse.jdt.core.prefs`. The dialogs offer some freedom in setting up the project. This flexibility is great when you're working on some project by yourself, but when there are multiple people working together, one developer may have a different project setup than the next, and so bugs are introduced. That's why we will use `gradlew` to generate a standard project setup for us:

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

Opening Xenon in Eclipse

After the Eclipse files have been generated, start Eclipse by typing the Bash alias `miniclipse` at the command line. From Eclipse's menu, select File→Import. In the Select dialog, select Existing projects into Workspace, then click the button labeled Next.

In the next dialog (Import projects), use the Browse... button to select the project's root directory, e.g. `/home/daisycutter/Xenon-examples`, then click Finish. An item Xenon-examples should now be visible in the Package explorer pane. Expand it, and navigate to `src/main/java/`, then select DirectoryListing.java from the nl.esciencecenter.xenon.examples.files package.

Right-click DirectoryListing.java and select Copy, then right-click again and select Paste. Eclipse should suggest the filename DirectoryListing2.java. Accept it. Now we have a file that we can play around with.

Double-click DirectoryListing2.java to bring up the corresponding Java code in the editor pane.

Running a Java program in Eclipse

So now that we have the source code open in the editor, let's see if we can run it. You can start the program in a couple different ways. For example, you can select Run→Run; you can use the key binding Ctrl+F11, or you can press the 'Play' icon in Eclipse's GUI. If you try to run the program, however, you will get the error we saw earlier at the command line (Eclipse prints the program's output to the pane labeled Console):

```
Example requires a URI as parameter!
```

So somehow we have to tell Eclipse about the URI (including both its scheme and its path) that we want to use to get to the contents of a directory of our chosing. You can do this through so-called 'Run configurations'. You can make a new run configuration by selecting Run from the Eclipse menu, then Run configurations.... In the left pane of the dialog that pops up, select Java Application, then press the New launch configuration button in the top left of the dialog. A new run configuration item should now become visible under Java Application. By default, the name of the run configuration will be the name of the class, but you can change the name to whatever you like. When you select the DirectoryListing2 run configuration in the left pane, the right pane changes to show the details of the run configuration. The information is divided over a few tabs. Select the tab labeled Arguments.

You should see a field named Program arguments where you can provide the arguments that you would normally pass through the command line. Earlier, we passed the string `file:$PWD`, but that won't work here, since `$PWD` is a Bash environment variable, and thus not directly available from within Eclipse. Eclipse does provide a workaround for this by way of the `env_var` variable. `env_var` takes exactly one argument, namely the name of an environment variable, such as `PWD`. The correct text to enter into Program arguments thus becomes `file:${env_var:PWD}`.

Debugging a Java program in Eclipse

In my opinion, one of the most helpful features of the Eclipse interface is the debugging/inspecting variables capability. This lets you run your program line-by-line. To start debugging, you have to set a breakpoint first. Program execution will halt at this point, such that you can inspect what value each variable has at that point in your program. Setting a breakpoint is most easily accomplished by double-clicking the left margin of the editor; a blue dot will appear. Alternatively, you can press Ctrl+b to set a breakpoint at the current line.

Set a breakpoint at the line

```
Xenon xenon = XenonFactory.newXenon(null);
```

Now that we have a breakpoint, we can run the program up to the position of the breakpoint. There are various ways to start a debug run: e.g. by selecting Run→Debug; or by pressing F11.

Run `CreatingXenon` up to the `xenon Xenon` line. Eclipse's Variables pane should now list just one variable, `args`, but since we supplied no arguments to `CreatingXenon`, `args` is currently a `String` object of length zero.

Press F6 to step over (and evaluate) the `xenon Xenon` line. You'll see a new variable `xenon` of type `XenonEngine` appear in the Variables pane. Expand the object to inspect it in more detail.

When you're done inspecting, press F8 to make Eclipse evaluate your program, either up to the next breakpoint, or if there are no breakpoints, up to the end your program.

Finally, you can terminate a debug run by pressing Ctrl+F2. Table 3.1 summarizes some of the most commonly used Eclipse key bindings used in running and debugging Java programs.

Table 3.1: Default key bindings used for running and debugging Java programs in Eclipse.

Default key binding	**Description**
F5	Step in
F6	Step over
F7	Step return
F8	Continue to the next breakpoint
F11	Start a debug run
Ctrl+F2	Terminate a debug run
Ctrl+b	Set a breakpoint at the current line
Ctrl+F11	Start a (non-debug) run

Chapter 4

Unused texts

4.1 Texlive

If you want to (modify and) build this document from its source, you'll need Texlive.

Texlive can be installed using

```
sudo apt-get install texlive
sudo apt-get install texlive-latex-extra
sudo apt-get install texlive-font-utils
```

4.2 Findbugs

Static code analysis with FindBugs

The FindBugs plugin for Eclipse lets you analyze your Java code for bugs. FindBugs is particularly useful when you are relatively new to Java, because it provides feedback on what goes wrong in your code (a 'bug pattern' in FindBugs terminology), and makes suggestions about how to change it.

In Eclipse's menu go to Help and select Install New Software... once more. Click Add.... For Name, type 'FindBugs'; for Location, type `http://findbugs.cs.umd.edu/eclipse`, then click OK. Make sure FindBugs is checked in the list, before clicking Finish. Restart Eclipse when prompted.

After the restart, you should be able to perform static code analysis. To do so, right-click the name of your project in the Package Explorer pane, there should be an item Find Bugs. Click it to start analyzing your code. After the analysis is done, you can view the results as follows: Go to Window→Show View→Other.... Look for the item labeled FindBugs and expand it. Select Bug Explorer and Bug Info. This should give you two extra panes with information of what type of bugs are in your code, where each bug is located, what makes it a bug, and what could be done to resolve it.

4.3 Docker

Later on in this document, we will take a closer look at continuous integration testing. Usually, setting up a testing environment is reasonably easy, but for Xenon it's a little more complicated. This is because the Xenon library is about connecting to remote systems, but in a testing environment, such remote systems do not exist—there's only the test machine. So, in order to run Xenon tests, we need to set up an environment in which a *virtual* remote system is used. Multiple virtual remote systems may in actual fact run on one *physical* machine. For now, let's just install the Docker[1] software that we'll use to set up virtual remote systems. Note that Docker needs a 64-bit host system. Also, it needs a minimum kernel version of 3.10 (again, on the host).

Check your kernel version with:

```
uname -r
```

Mine is `3.13.0-67-generic`.

The Ubuntu repositories contain an older version of Docker, which you should not use. Instead, use the newer version from Docker's own PPA.

First check if you have the older version by:

```
docker -v
```

Mine says:

```
Docker version 1.9.0, build 76d6bc9
```

[1]The remainder of this section is based on: `https://docs.docker.com/engine/installation/ubuntulinux/`

If your version is lower, go ahead and uninstall as follows. First find out where your Docker program lives with:

```
which docker
```

and then find out which package your Docker is a part of with:

```
dpkg -S `which docker`
```

If you already had Docker installed, then the package name is likely either `docker.io` or `lxc-docker`. Either way uninstall the entire package, including its settings with

```
sudo apt-get remove --purge docker.io
```

or

```
sudo apt-get remove --purge lxc-docker*
```

We will use software from a third-party repository, `https://apt.dockerproject.org`. For this, we'll need to add the new repository's PGP key to our installation as follows:

```
sudo apt-key adv --keyserver hkp://pgp.mit.edu:80 \
--recv-keys 58118E89F3A912897C070ADBF76221572C52609D
```

The details of the next step vary depending on the operating system you are using, so let's first check which version you are running:

```
lsb_release -dc
```

Make a note of your distribution's codename for the next step (mine is `trusty`).

Open or create the file `/etc/apt/sources.list.d/docker.list` in an editor such as nano, gedit, leafpad, etc. I'm using nano:

```
sudo nano /etc/apt/sources.list.d/docker.list
```

Delete any existing entries in `/etc/apt/sources.list.d/docker.list`, then add one of the following options

```
deb https://apt.dockerproject.org/repo ubuntu-precise main
deb https://apt.dockerproject.org/repo ubuntu-trusty main
deb https://apt.dockerproject.org/repo ubuntu-vivid main
deb https://apt.dockerproject.org/repo ubuntu-wily main
```

(I chose the second because I'm on trusty).

Next, save and close `/etc/apt/sources.list.d/docker.list`.

Now that we have added Docker's PPA to the list of software sources, we need to update the list with the package information as follows:

```
sudo apt-get update
```

Check if your are now using the right docker:

```
apt-cache policy docker-engine
```

Mine says:

```
docker-engine:
Installed: 1.9.0-0~trusty
Candidate: 1.9.0-0~trusty
Version table:
***1.9.0-0~trusty 0
     500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
     100 /var/lib/dpkg/status
   1.8.3-0~trusty 0
     500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
   1.8.2-0~trusty 0
     500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
   1.8.1-0~trusty 0
     500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
   1.8.0-0~trusty 0
     500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
   1.7.1-0~trusty 0
     500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
   1.7.0-0~trusty 0
     500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
   1.6.2-0~trusty 0
     500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
   1.6.1-0~trusty 0
     500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
   1.6.0-0~trusty 0
     500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
   1.5.0-0~trusty 0
     500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages
```

Now for the actual install. If your Ubuntu version is Ubuntu Wily 15.10, Ubuntu Vivid 15.04, or Ubuntu Trusty 14.04 (LTS), you're in luck, as these OS'es have everything you'll need already. If you're not on one of these Ubuntu versions, refer to `https://docs.docker.com/engine/installation/ubuntulinux/` for instructions on installing some additional packages before proceeding with the next step.

Install Docker with:

```
sudo apt-get install docker-engine
```

The Docker service should have started; if for some reason it hasn't, you can start it manually by:

```
sudo service docker start
```

Now let's try a small example to see if Docker works:

```
sudo docker run hello-world
```

This command downloads a test image `hello-world` from DockerHub, an external repository for storing Docker images. Just to be clear, an 'image' in this context refers to an image of an operating system—it has nothing to do with a picture.

When the container runs, it prints an informational message. Then, it exits.

You can check where docker images are stored by:

```
docker info
```

Mine are stored under `/var/lib/docker`; whatever the location, make sure you have enough disk space there, as Docker will download any new containers to that location.

The Docker daemon binds to a Unix socket instead of a TCP port. By default that Unix socket is owned by the user `root` and other users can access it with `sudo`. For this reason, the Docker daemon always runs as the `root` user.

To avoid having to use `sudo` when you use the `docker` command, we will create a Unix group called `docker` and add users to it. When the Docker daemon starts, it makes the ownership of the Unix socket read/writable by the `docker` group.

Add yourself to the `docker` group with:

```
sudo usermod -G docker -a <name-of-user>
```

Log out and back in.

We will use multiple Docker containers simultanenously. To coordinate how individual Docker containers talk to each other, we need a tool called `docker-compose`[1]. It uses a so-called compose file to configure an container's services. Xenon's compose file is `docker-compose.yml` located in `src/integrationTest/docker/`.

To install the `docker-compose` program, first check `https://github.com/docker/compose/releases` to see what the latest stable version of `docker-compose` is. This determines the `VERSION_NUM` in the command below. Mine is `1.5.0`.

Download `docker-compose` using `curl` form the terminal:

```
cd ~
curl -L https://github.com/docker/compose/releases/downlo\
ad/VERSION_NUM/docker-compose-`uname -s`-`uname -m` > docker-compose
```

Then move the downloaded file into the right directory on your system with:

```
cd ~
sudo mv docker-compose /usr/local/bin/
```

Apply executable permissions to the binary:

```
sudo chmod +x /usr/local/bin/docker-compose
```

[1]For more information on installation, see: `https://docs.docker.com/compose/install/`

And verify that it worked:

```
docker-compose --version
```

Mine says:

```
docker-compose version: 1.5.0
```

That's it for now, we will get back to Docker in Section **??**: **??**.

4.4 Gradle

In order to use the source code from the library, we need to compile it first. Compiling source code can be a repetitive task, but fortunately there are tools available that automate much of the build process. Xenon uses Gradle for building the library.

Install Gradle using

```
sudo apt-get install gradle
```

How Gradle uses conventions: an example

In this section, I'll try to demonstrate how little configuration Gradle needs in order to be able to build Java projects successfully—as long as you don't deviate from existing conventions with regard to how the code is organized.

Make a new directory, for example ~/`tmp/hellogradle`:

```
cd ${HOME}
mkdir -p tmp/hellogradle
```

Then `cd` into the new directory and make a new subdirectory `src/main/java` in it. `src/main/java` is the conventional place where Java source code is stored.

Now open a text editor and copy-paste this Java class into it:

```
package nl.esciencecenter.hellogradle;

public class HelloGradle {

    public static void main(String[] args) {
        System.out.println("Hello, Gradle");
    }

}
```

And then save the Java class as `HelloGradle.java` in the newly created `src/main/java` directory. Files containing Java source code should have file-

names that end in `.java`.

Now let's make the simplest Gradle build file. Open a new text editor and copy-paste this into it:

```
apply plugin: 'java'
```

Then save as `build.gradle` (the default name for Gradle build files) in the `~/tmp/hellogradle` directory. At this point, `~/tmp/hellogradle` should contain this:

```
build.gradle
src
src/main
src/main/java
src/main/java/HelloGradle.java
```

The `apply plugin` line in `build.gradle` tells Gradle to use the Java plugin for Gradle. The Java plugin defines a number of build tasks typical for a Java project. It also specifies the default locations for the project's Java source code (`src/main/java`), the compiled Java classes (`build/classes`), the documentation (`build/docs/javadoc`), as well as some other things that we'll check out later, such as unit testing and integration testing. You can find the details online at `https://docs.gradle.org/current/userguide/java_plugin.html`.

Gradle can tell you what tasks it knows about. Try running

```
cd ${HOME}/tmp/hellogradle
gradle --build-file build.gradle tasks
```

The `--build-file` option is used to specify which file to use as input to gradle; its short version is `-b`. This will give you the default tasks that Gradle knows about, such as `help` and `properties`. Note that `tasks` itself is a task, and so it is listed along with the other tasks. Additionally, it will list any tasks that are defined in `build.gradle`. Note that this includes any tasks which are defined implicitly (for example by `apply plugin` lines). To get a little more information on each task's dependencies, you can add the `--all` option as follows:

```
cd ${HOME}/tmp/hellogradle
gradle --build-file build.gradle tasks --all
```

In the output, there should be a task called `classes`, which compiles the (main) source code. The task `classes` is added by the Java plugin. Let's build our `src/main/java/HelloGradle.java` and see what that gives us. Run:

```
cd ${HOME}/tmp/hellogradle
gradle --build-file build.gradle classes
```

Afterwards, the directory should contain the following files and directories:

```
build
build/classes
```

```
build/classes/main
build/classes/main/nl
build/classes/main/nl/esciencecenter
build/classes/main/nl/esciencecenter/hellogradle
build/classes/main/nl/esciencecenter/hellogradle/HelloGradle.class
build/dependency-cache
build.gradle
src
src/main
src/main/java
src/main/java/HelloGradle.java
```

As you can see, Gradle generated a directory `build` and put all the things it built in it. It created a subdirectory `classes`, containing yet another subdirectory `main`, since that is the name of the so-called 'sourceSet' (a collection of source code files that belong together conceptually). Inside `main`, there are nested subdirectories for each part of the package name `nl.esciencecenter.hellogradle`. Finally, there is the compiled Java class `HelloGradle.class`. Note that files containing compiled Java code should have filenames that end in `.class`.

Let's see if the compile worked. Run

```
cd ${HOME}/tmp/hellogradle
java -classpath build/classes/main nl.esciencecenter.hellogradle.HelloGradle
```

The `-classpath` option tells Java it should look in `build/classes/main` for compiled Java classes (its short option name is `-cp`). The last argument, `nl.esciencecenter.hellogradle.HelloGradle` is the fully qualified name of the class we want to run. If everything worked, you should see the 'Hello Gradle' greeting.

Javadoc

Java comes with a neat system of automatically documenting source code, called 'Javadoc'. Javadoc is able to parse the Java source code, analyzing its structure for things like class hierarchy, public interfaces, public class methods, constructors, etc. Javadoc then generates the corresponding documentation automatically. The great advantage of generating the documentation in an automated way *from the source code* is that the documentation is always up to date with how the code works.

Thanks to the Java plugin, Gradle knows how to generate Javadoc (if you run `gradle --build-file build.gradle tasks` again, you'll see a task `javadoc`, which as the name suggests, generates Javadoc documentation. You can run the `javadoc` task in the same way as you run any other task:

```
cd ${HOME}/tmp/hellogradle
gradle --build-file build.gradle javadoc
```

The `build` directory should now have a few new items:

```
build
build/docs
build/docs/javadoc
build/docs/javadoc/nl
build/docs/javadoc/nl/esciencecenter
build/docs/javadoc/nl/esciencecenter/hellogradle
build/docs/javadoc/nl/esciencecenter/hellogradle/package-summary.html
build/docs/javadoc/nl/esciencecenter/hellogradle/HelloGradle.html
build/docs/javadoc/nl/esciencecenter/hellogradle/package-frame.html
build/docs/javadoc/nl/esciencecenter/hellogradle/package-tree.html
build/docs/javadoc/deprecated-list.html
build/docs/javadoc/constant-values.html
build/docs/javadoc/allclasses-noframe.html
build/docs/javadoc/overview-tree.html
build/docs/javadoc/index.html
build/docs/javadoc/help-doc.html
build/docs/javadoc/index-all.html
build/docs/javadoc/stylesheet.css
build/docs/javadoc/resources
build/docs/javadoc/resources/background.gif
build/docs/javadoc/resources/tab.gif
build/docs/javadoc/resources/titlebar_end.gif
build/docs/javadoc/resources/titlebar.gif
build/docs/javadoc/allclasses-frame.html
build/docs/javadoc/package-list
build/tmp
build/tmp/javadoc
build/tmp/javadoc/javadoc.options
build/classes
build/classes/main
build/classes/main/nl
build/classes/main/nl/esciencecenter
build/classes/main/nl/esciencecenter/hellogradle
build/classes/main/nl/esciencecenter/hellogradle/HelloGradle.class
build/dependency-cache
build.gradle
src
src/main
src/main/java
src/main/java/HelloGradle.java
```

You can use any web browser to navigate through the Javadoc documentation (`build/docs/javadoc/index.html` is probably the best starting point).

Gradle also defines a task to clean up the directory, such that you only have the bare essentials. This task is called `clean` and can be run with

```
gradle --build-file build.gradle clean
```

Note that in order to generate the documentation, the source needed to be compiled first. The task `javadoc` is said to *depend on* the task `classes`. You can check that this is indeed the case by:

```
gradle --build-file build.gradle javadoc
```

This should generate `build/docs/javadoc` as before, but only after generating `build/classes` first.

So far, we've been explicitly specifying what build file Gradle should use through `gradle`'s `--build-file` option. However, as long as you are using the default build file (`build.gradle`), there's no need to be explicit about that—you can just run `gradle` followed by the name of the task, e.g.:

```
gradle javadoc
```

While Javadoc's automated documentation generation is helpful when it comes to the *how* of the Java code, it can provide little in terms of the *why* (or the *why like this*). The solution for that particular problem necessarily requires input from the programmer. That is, the programmer can clarify his/her Java code by including Javadoc directives (*tags*), which can help explain the meaning of input arguments, variables, methods, classes, and so forth. The most common tags are listed in Table 4.1.

Table 4.1: Commonly used Javadoc tags.

Tag name	**What the tag does**
`@author`	Specifies the author(s).
`@version`	Specifies the version.
`@param`	Describes a method parameter.
`@return`	Describes the (meaning of the) returned value.
`@throws`	Describes what exceptions are thrown by this method.
`@see`	Provides a link to another element of the documentation.
`@link`	References a place in the javadoc.
`@deprecated`	Advises the user not to use a program element anymore, and ideally specifies what to use instead.
`@since`	Specifies when certain functionality was first introduced.

Note that by convention the tags appear in this order, and that Javadoc tags are lowercase.

Personally, I think `@author` is probably best avoided—if you want to know who contributed what, ask the version management system. For example, if you have a file `UnknownPropertyException.java` and you use `git` for version management, entering `git blame UnknownPropertyException.java` at the command line will give you a line-by-line overview of who commited what, when. Online code repositories such as Github typically also provide this capability in the form of a *blame button*. You may use `@author unascribed` if you do not want to, or cannot specify the author.

As a small example, update `HelloGradle.java` with a Javadoc comment, explaining what this class does, as follows:

```
package nl.esciencecenter.hellogradle;

/**
 *  This is a short one-line description of the class.
 *  <p>
 *  The Javadoc comment block appears just before the class
 *  it describes.
 *  </p>
 *  <p>
 *  You could use another paragraph to explain more stuff.
 *  </p>
 *
 *  @author      Firs T. Author
 *  @author      Secon D. Author
 *  @version     1.0, 9-Sept-2015
 *  @deprecated  As of release 1.3, replaced
 *               by {@link nl.esciencecenter.ByeGradle}
 */
public class HelloGradle {

    /**
    * 'main' is the only method.
    */
    public static void main(String[] args) {
        System.out.println("Hello, Gradle");
    }

}
```

Now run `gradle javadoc` again and check how that changed `build/docs/javadoc/index.html`.

Note that Javadoc expects the commenting style to be exactly like this:

```
/**  <- javadoc opening tag
 *  (any number of lines like this line)
 */  <- javadoc closing tags
```

The Gradle wrapper

So far, we've been calling the `gradle` executable directly. There is, however, a better way of starting a Gradle build, namely by using the so-called 'Gradle wrapper'. The Gradle wrapper consists of a shell script (`gradlew`), a configuration file (`gradle/wrapper/gradle-wrapper.properties`), and some compiled Java code bundled into a Java archive (`gradle/wrapper/gradle-wrapper.jar`).

The Gradle wrapper is the preferred way of starting a Gradle build. This is because using `./gradlew` offers a couple of advantages over using `gradle`. Firstly, users don't need to install Gradle in order to build the software. This

is particularly convenient when building the software on machines that are not owned or maintained by the user, as is typically the case during *continuous integration testing* using software such as Jenkins or Travis; we'll take a look at testing later. Secondly, using the Gradle wrapper gives you the option of running a Gradle version which is more up-to-date than what's available in the operating system's software repositories. For example, my Lubuntu 14.10 comes with Gradle version 1.4, while Gradle is currently at version 2.7. Thirdly, since the Gradle wrapper files are checked into the version control system, they become part of the software. This ensures that software users are running the exact same build setup as are the software developers, which improves the robustness of the software when everything is running smoothly, and improves reproducability when bugs occur.

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
```

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 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.

4.5 Gradle

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 Maven-Central[2] (a website that hosts many common Java packages, in many different

[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.

[2] `https://repo1.maven.org/maven2`

versions); if the package is not available from MavenCentral, or if the download fails for some other reason, Gradle tries a different website (Bintray[1]). 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`
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.

[1] `https://bintray.com/bintray/jcenter`

Index