Distributed Computing with Xenon
tutorial

Jurriaan H. Spaaks

December 14, 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

Xenon consists of three pillars: 'credentials', 'files', and 'jobs'. The credentials pillar contains functionality pertaining to credentials. Credentials (such as a name and password combination) are often required to gain access to files or to submit jobs. The files pillar contains all functionality relating to files, such as creation, deletion, copying, reading, writing, obtaining a directory listing, etc. Lastly, the jobs pillar contains functionality relating to jobs, e.g. submitting, polling, cancelling, etc.

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

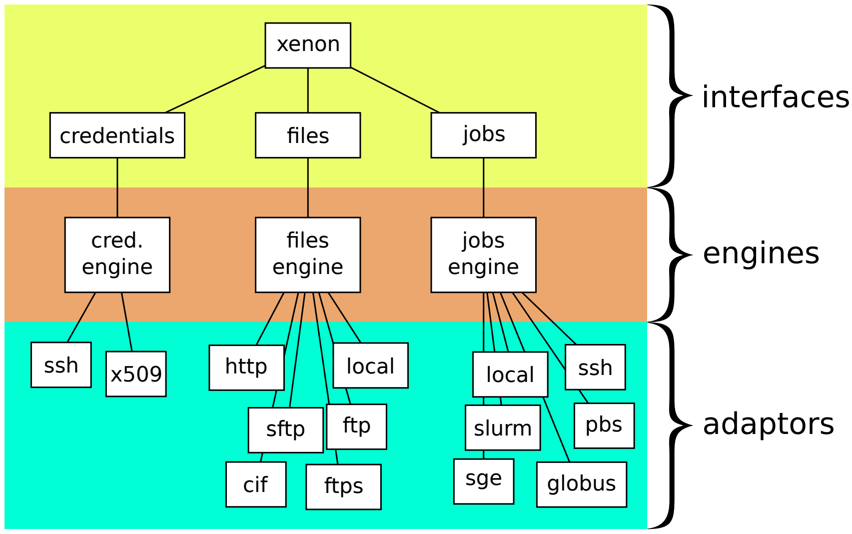

Figure 1.1: Xenon is built on 3 pillars: 'credentials', 'files' and 'jobs'. Each pillar consists of an interface, an engine, and an adaptor.

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 Building with `gradlew`

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

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-examples
./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

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

If you run `./gradlew tasks`, you'll see a line at the top that says that the

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

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.

2.4 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-examples
java nl.esciencecenter.xenon.examples.files.DirectoryListing
```

you will get the error below:

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

```
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-examples`, our classpath thus becomes `build/libs/Xenon-examples-all.jar`. However, if we now try to run

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

If all goes well, you should now see the contents of the current directory as an INFO message.

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

2.5 Setting the log level

Xenon uses the logback[1] library for much of the output. For this, developers have sprinkled the code with so-called *logger* statements that produce messages. Each message has been annotated as an error, a warning, debugging information, or as a plain informational message. The logback library lets the user configure where each type of message should be routed. For example, warnings and informational messages may be routed to standard output, while error messages may be routed to standard error. Furthermore, it gives the user control of how each message should be formatted, for example with regard to what class produced the message, and at what line exactly. The behavior of the logger can be configured by means of a file called `logback.xml`. It is located at `src/main/resources/`. There should not be any need for you to change `logback.xml` too much, but if you do, make sure to re-run `./gradlew` for your changes to take effect.

By default, `logback.xml` uses a logging level of `INFO`, which means that warnings, errors, and informational messages are routed to standard output, but debug messages are not visible. At some point, you may find yourself in a situation where you want to change the logging level. There's two ways you can do that: first you can change the default behavior by editing the `loglevel` line in `src/main/resources/logback.xml`, saving it, and re-running `./gradlew`.

[1] `http://logback.qos.ch/`

Secondly, you can keep the defaults as they are, but only run a specific call with altered `loglevel` settings, by using command line parameters to the `java` program. For example, you could lower the logging level from `INFO` to `DEBUG` as follows:

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

Standard output will now include messages of the `WARN`, `ERROR`, `INFO`, and `DEBUG` level:

```
time   : 16:52:38.393 (+271 ms)
thread : main
level  : DEBUG
class  : nl.esciencecenter.xenon.adaptors.ssh.SshAdaptor:179
message: Setting ssh known hosts file to: /home/daisycutter/.ssh/config

time   : 16:52:38.396 (+274 ms)
thread : main
level  : WARN
class  : nl.esciencecenter.xenon.adaptors.ssh.SshAdaptor:242
message: OpenSSH config file cannot be read.

time   : 16:52:38.412 (+290 ms)
thread : main
level  : INFO
class  : nl.esciencecenter.xenon.engine.XenonEngine:169
message: Xenon engine initialized with adaptors: [Adaptor [name=local], Ada...
```

2.6 `ssh` to `localhost`

Up till now, you've used Xenon to perform tasks on your own system (by means of Xenon's `local` adaptor). Usually though, you'll want to use Xenon to perform tasks on remote systems, such as the Amazon cloud, the Lisa cluster computer, the Cartesius supercomputer, or perhaps just a single machine (but not your own). In order to do anything on those remote systems, you first need to connect to them. On Linux, you can use the so-called *secure shell* program `ssh`. With `ssh`, you can set up an SSH connection to a remote system. Through SSH, you can do all the things a user is allowed to do on that system (which in turn is defined by the remote system's administrator). Note that SSH connections are a crucial part of how Xenon works: without them, you can not do anything that requires remote resources.

SSH uses the server-client model. Normally, your system is the client, while the remote system is the server. However, let's first see if you can connect to your own system using `ssh`. Your system will thus play the part of both client and server.

The client needs to have a so-called *public-private key pair*. The key pair can be generated according to different algorithms: `dsa`, `ecdsa`, `ed25519`, or `rsa`. The Linux program `ssh-keygen` implements these algorithms; we will use it to generate a public-private key pair using the `rsa` algorithm, as follows:

```
# generate key pair using rsa algorithm
ssh-keygen -t rsa
# (press enter to accept the default file location ~/.ssh/id_rsa; if it
# tells you the file already exists, you can skip creation of a new key)
# (press enter to set the passphrase to an empty string)
# (press enter to confirm setting the passphrase to an empty string)
```

You should now have a directory ~/`.ssh` with two files in it: `id_rsa`, the private part of the key pair and `id_rsa.pub`, the public part of the key pair. Note that the contents of `id_rsa` should remain a secret.

That's it for the client side, but you still need to do some stuff on the server side. First, install the package `openssh-server` from the Ubuntu repositories:

```
# install a server
sudo apt-get install openssh-server
```

The server side keeps a list ~/`.ssh/authorized_keys` of trusted identities. You have to add the identity information from ~/`.ssh/id_rsa.pub` to it. Linux provides an easy way of doing this, as follows:

```
# copy identity, type 'yes' when prompted
ssh-copy-id ${USER}@localhost
```

On the server side, file ~/`.ssh/authorized_keys` should now contain the information from ~/`.ssh/id_rsa.pub`.

On the client side, SSH tried to connect the current user (whose credentials are found in ~/`.ssh/id_rsa.pub`) to the 'remote' system `localhost`. If that was successful, there should be a new file ~/`.ssh/known_hosts`. `known_hosts` is a list of machines that we have connected to previously; i.e. these are systems we trust.

The structure of `known_hosts` is as follows. Each line begins with the following elements:

- `1` a flag. Here, it signifies that the third element (host name) is hashed using the SHA1 algorithm;
- `rb6bKm69i0R/885WKLX7HZR0CjY` the (public) salt used to encrypt the host name;
- `NoA7R+Ivcef4+rtCLYqBMlqPT+A` the (hashed) host name;
- key-value pairs, e.g. the key `ecdsa-sha2-nistp256` and its value

`AAAAE2V...<characters omitted>...RpXi/rE`, representing the public key of the 'remote' system which was generated when we installed `openssh-server` and which is stored at `/etc/ssh/ssh_host_ecdsa_key.pub`[1].

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

```
# now try to log in without entering key:
ssh ${USER}@localhost
```

2.7 DirectoryListing through ssh

So now that we have configured passwordless `ssh` to `localhost`, listing the contents of a directory on `localhost` is actually pretty simple. In the URI that you pass as a command line argument to `DirectoryListing` (see page 8), replace `local` by `ssh`, and supply `localhost` as the domain name. Also note that the inclusion of a domain makes it necessary to include `//` between `ssh` and `localhost` (see footnote on formatting URIs on page 8).

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

If all goes well, you should see the same directory contents as before, but now viewed through SSH.

[1] You can show the fingerprint of the server's public key file using:
`ssh-keygen -l -f /etc/ssh/ssh_host_ecdsa_key.pub`
The number that `ssh-keygen` returns should be the same number that was displayed in your terminal when you did `ssh-copy-id $USER@localhost`.

2.8 Example usage of `local` and `ssh` adaptors

We have compiled a list of examples demonstrating various uses of the `local` adaptor and the `ssh` adaptor. Your system is now configured correctly to run all of them. This is most easily accomplished by starting the `run-examples.sh` script we have prepared:

```
cd ${HOME}/Xenon-examples
./run-examples.sh
```

Note that you can edit the loglevel by changing the parameter `LOGLEVEL` in `run-examples.sh`.

For now, we'll just use SSH to connect to our own machine, but later on in this document we'll also look at using SSH to connect to physically remote systems, such as the Lisa[1] and DAS-4[2] systems (see Section 3.7: *Live systems*). Additionally, we'll set up Docker to run virtualized systems on your local machine, with which you can emulate the behavior of one or more remote machines (see Section 4.1: *Docker*). Anyway, that's all for later—first we'll make a small digression into setting up the Integrated Development Environment or IDE called Eclipse.

[1] `https://userinfo.surfsara.nl/systems/lisa`
[2] `http://www.cs.vu.nl/das4/`

Chapter 3

Eclipse

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

3.1 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 to. 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 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).

3.2 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-examples
./gradlew eclipse
```

3.3 Opening the Xenon examples 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 double-click DirectoryListing.java from the nl.esciencecenter.xenon.examples.files package to view its code in the editor pane.

3.4 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):

```
time   : 18:42:04.689 (+219 ms)
thread : main
level  : ERROR
class  : nl.esciencecenter.xenon.examples.files.DirectoryListing:51
message: 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 DirectoryListing 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 `local:${PWD}`, but that won't work here, since `${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 `local:${env_var:PWD}`. Note that `${env_var:PWD}` refers to the directory that Eclipse was started from.

3.5 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+Shift+b to set a breakpoint at the current line.

Set a breakpoint at the line

```
URI uri = new URI(args[0]);
```

Now we need to set the debug configuration in a similar manner as we did for the run configuration. Select Run from the menu, select Debug configurations... (not Run configurations...), then select the configuration we used previously.

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.

You can add all kinds of helpful tools to the Eclipse window; for an overview of your options, click the Window menu item, then select Show view, then select Other.... Select whatever tools you like, but make sure to select the Variables tool from Debug. This tool allows you to view information about your variables while you're debugging. Use drag and drop to lay out the Eclipse window to suit your needs. Eclipse refers to its window layout as a 'perspective'; perspectives can be saved by subsequently selecting Window, Perspective, and Save perspective as.... This allows you to have custom perspectives for development in different languages (Java, Python, C, etc.), or for different screen setups (laptop screen v. side-by-side 1920x1080 for example), or for different tasks (Java development v. Java debugging for example).

Getting back to DirectoryListing, execution has been halted just before the line `URI uri = new URI(args[0]);` was executed. If you now look in the Variables tool pane, there should be only one variable visible: `args`, which contains the string we supplied through the Program arguments of the debug configuration.

Press F6 to evaluate the line. You'll see a new variable `uri` of type `URI` 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 common 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+Shift+b	Set a breakpoint at the current line
Ctrl+F11	Start a (non-debug) run

3.6 Setting the log level

Earlier, we looked at how to pass custom `loglevel` values to `java` using command line parameters such as `-Dloglevel=DEBUG` (see section 2.5). Passing command line parameters is also possible in Eclipse, by altering the debug configuration (or run configuration) as follows. In Eclipse's menu, go to Run and select either Run configurations... or Debug configurations... as appropriate. Then, in the left pane, select the Java application whose configuration you want to adapt. In the right-hand pane, subsequently select the tab labeled Arguments. The second text field from the top should be labeled VM arguments. Here you can add command line parameters to the java program, such as `-Dloglevel=DEBUG`.

3.7 Live systems

Earlier, we set up passwordless `ssh` to connect to `localhost`. Obviously, using `ssh` to connect to your own system is a bit silly. Normally, you'll want to connect to a physically remote system. In the next two sections, I'll explain how to connect to SURFsara's cluster computer, Lisa, and VU University's DAS-4 cluster, respectively.

3.7.1 `ssh` to SURFsara's Lisa cluster computer

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`.

First, I verify that I can connect to Lisa's head node, using the `ssh` command below:

```
# (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 `~/.ssh/known_hosts` now includes a line for the Lisa system.

[1]SURFsara publish RSA public 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.

3.7.2 `ssh` to DAS-4 cluster computer

TODO

Chapter 4

Unused texts

4.1 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.

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

First check if you have the older version by:

```
docker -v
```

Mine says:

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

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`.

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

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
```

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 **??**: **??**.

Index