

## Appendix 1. The Java ANT utility: Managing the production of programs

Computer applications encompass manyfold software elements, not the least their core of programs themselves, but also libraries, configuration files and the versions thereof. Managing their production processes and automating their documentation entails using an advanced software tool, a *build tool*. Included in software development environments, build tools help programmers manage the numerous parameters and procedures that are tailored to each specific build.

In addition, distributed applications entail a build tool that is itself distributed so that construction and deployment can be achieved in coordination with other build tools or with other remote software. There are a bunch of build tools available today, some of them are tailored to specific development environments, some others are more focused on specific types of applications. The ANT build tool, which we present here, is based on Java and is specialized for Java development, and offers features that make it a good fit for distributed development. ANT facilitates Java development, since it is a Java application. However, it doesn't preclude other languages, or a mix thereof. In this tutorial, we present ANT with the aim of making easier doing those programming practices from this book which are written in Java. We include a forthcoming tutorial similar to this one for presenting the MAKE build utility, the legendary build tool included in the AT&T original invention of the UNIX operating system which original aim was development in C and Bourne Shell scripting.

### Guided overview of ANT

Developing a *simple* C/S application will enable us to acquaint ourselves with ANT. The technical intricacies to the application itself are explained with detail in the lesson on Distributed Objects; for the time being though, we'll be contented with focusing on ANT and will take for granted the software design itself.

The application's server (S) program exports a remote object that offers two remote methods. Those methods are to be called by the application's client (C). As is mandatory in Java RMI, the same host that executes the server must also run at least an instance of Java's platform program *rmiregistry*. This program stores a copy of the server's proxy object, which will be used later by the client as a *stand-in* object representing the remote object. Thus, the client, by calling the methods of the stand-in object, ultimately will be able to invoke the remote object's methods. The diagram in fig. 100 captures these essential Java RMI elements.

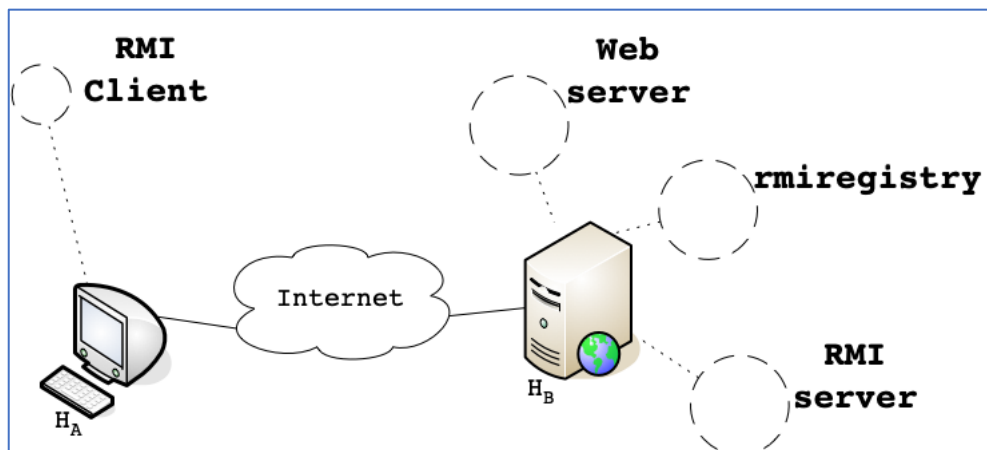


Figure 100. Overall structure of the example project

Java ANT helps us manage the compilation, deployment and execution of the server and the client, the launching of the server host's rmiregistry and the uploading of the server's remote interface to a web server accessible to the client. Java ANT, can be configured to take care of the dependencies among the different software elements mentioned above. That is great help whenever runtime errors happen as the foreseeable, endless repetitions of the development cycle will be enacted, since each will involve numerous software elements, running in separate JVMs which may be run in separate hosts.

Simply skimming the text below, which describes the example project will set us off taking stock of Ant without the need to have it installed beforehand. After studying the example's process of software construction and deployment, if you deem it convenient for you, you can proceed to the actual installation of Ant. At that moment you'll be able to fully test the software construction and deployment under your own infrastructure.

When you finish skimming this example project deployment, you'll be more familiar with ANT, and that will help you grasp the essential concepts and possibilities of ANT more easily. It figures that this tutorial requires Java and the Java Compiler installed in your system and that we assume Linux as the operating system, despite that being true, we think that the software that we are illustrating here will function on OS-X, Unix with only minor modifications, if any. Follow this outline of steps to fully deploy and run the *distributed* project:

1. Open an ssh session in host H<sub>B</sub> or login into that host directly from its console, for example:

```
$ ssh your\_login\_name@192.168.1.99  
...
```

2. Download the project's source files. Use a web browser or an http download utility such as Linux's wget or OS-X curl to download the zip file that contains the source code:

```
$ wget http://paloalto.unileon.es/ds/lab/rmipract.zip
```

Check file `rmipract.zip` was successfully downloaded:

```
$ ls -l
total 16
-rw-r--r-- 1 networks networks 12678 Nov  8 10:32 rmipract.zip
```

3. Decompress the `.zip` file by using an appropriate decompressor like `unzip`:

```
$ unzip rmipract.zip
```

Check the resulting file tree with the Linux `tree` command or a similar one:

```
$ tree
.
├── build
├── build.xml
├── __MACOSX
│   └── src
│       ├── client
│       └── server
├── rmipract.zip
└── src
    ├── all.policy
    ├── client
    │   ├── all.policy
    │   └── Client.java
    └── server
        ├── SDRemoteObjectImpl.java
        ├── SDRemoteObject.java
        └── Server.java

8 directories, 8 files
```

Directory `__MACOSX` is relevant to OS-X file browser, however, we'll leave it as is. The remaining directories have this import here:

- `src`: This directory contains project sources and configuration text files, organized into subdirectories that represent the *Java package* structure of the project.
- `build`: The class, jar files and others resulting from the compilation and linking of the project will be saved here.
- `build.xml` file: This file contains the instructions ANT will apply when forming the project build. The format of this file is xml, the *xml markup language*. All the elements relevant to ANT, such as properties, targets, external actions, libraries, etc. are specified within this file. The specifications are formatted within the included xml tags that ANT understands and which carry its semantics.

4. XML text file `build.xml` contains the sequences of instructions that ANT will follow to *build* the various target libraries, classes and other types of files that comprise the final project runtime. In addition, file `build.xml` carries instructions for deploying the runtime along its support programs (`rmiregistry`, for example) and for orderly executing each. If this deployment requires file uploading and downloading at remote systems, ANT supports execution of the ssh-based `scp` utility (The secure copy utility is based on the Secure Shell suite of protocols and applications. Check out its man page if you need to see more detail).

Overall, ANT's build file contains the following sections. You can skim the file's actual contents as the explanations unfold, below. The build file is formatted in XML (Extensible Markup Language), consequently, you'll immediately notice that it is composed of a number of XML tags specific to ANT and others which are relevant and specific to our concrete project under development:

- **Project name:** The project name is included in an ANT's XML tag which name is *project*. It is used for marking the start section of ANT's XML build file:

```
<project name="DSRmi" basedir="." default="run_server">
```

The closing tag is : `</project>`

- **Properties:** This section allows creating symbolic strings that will help name the different elements that constitute the project in a manner practical, simple customized to the programmer's. For example, the following line creates a symbolic string that will be used for identifying the IP address of a web server relevant to the project:

```
<property name="webserver.ip" value="192.168.1.99"/>
```

**Targets:** Each target specifies a procedure that will be executed by ANT whenever the user commands it to and which will lead to the production of one of the project's elements such as a library, a class file, a jar file, etc. In the following example about target "compile\_server", a server program is compiled and a certain file required by RMI is copied to a web directory. Compiling occurs first; file copy occurs next, as expressed by the ordering of the actions within the target's tag:

```
<target name="compile_server" depends="clean_server">
    <javac
        srcdir="${server.src.dir}"
        destdir="${server.dir}">
        <compilerarg value="-encoding" />
        <compilerarg value="utf8" />
    </javac>
    <copy file="${server.dir}/${package.name}/SDRemoteObject.class"
        todir="${webdir}/${package.name}"/>
```

```
</target>
```

The tag `<javac>` is used for executing the Java compiler command-line utility `javac`. The parameters values passed to it, such as `srcdir`, `destdir` are taken from the properties ANT tag, at the beginning of the build file, where they are initialized.

The `<copy file .../>` tag included on the last section of target “`compile_server`” copies the stub class file to an appropriate destination directory.

The `depends` argument `depends=“clean_server”` passed in the target tag specifies that whenever the user executes this target, target “`clean_server`” should always be executed before.

Skim the contents of file `build.xml` and check that you can identify the main sections listed above. You may use the Linux `more` command, for example:

```
$ more build.xml
```

5. Compile the server. Let's begin by compiling the server, and let's do so not by invoking the `javac` command, but by having `ant` do that by fetching the right *ant target* from file `build.xml`. The relevant `ant target` is “`compile_server`” which you just skimmed. Compose the following command:

```
$ ant compile_server
Buildfile: /home/networks/book/build.xml

clean_server:
  [mkdir] Created dir: /home/networks/book/build/server

compile_server:
  [javac] /home/networks/book/build.xml:58: warning:
'includeantruntime' was not set, defaulting to
build.sysclasspath=last; set to false for repeatable builds
  [javac] Compiling 3 source files to
/home/networks/book/build/server
  [copy] Copying 1 file to /var/www/html/dsrmipract

BUILD SUCCESSFUL
Total time: 0 seconds
$
```

Notice that `ant` launched target “`clean_server`” first, which is in accordance with the *depends* instruction included in the target that was meant to be executed, “`compile_server`”. Check the text to that target if you wish. Then, “`compile_server`” is itself launched which runs `javac` first and then the file copy next to it.

The copied file is the compiled Java interface of the remote object exported by the server. This compiled Java interface is dubbed *a stub* and is required so that `rmiregistry` can enact its service of storing remote object proxies. The *stub* has been compiled within the same target “`compile_server`”.

All these operations are reported to be successful. You may disregard the warning altogether for the purposes of this overview.

We won't run the server yet, even though we successfully compiled it. The server, *i.e.*, the RMI Server requires an instance of `rmiregistry` running in the same host where the server is being run, consequently, we must launch `rmiregistry` before deploying and running the server. Again, we'll resort to `ant`; this time the target is “`rmiregistry`”.

6. Launch `rmiregistry`. The relevant target is “`rmiregistry`” and the action it will launch whenever we invoke it from the command line is `<exec ...>`. The Linux command passed as argument to the `<exec ...>` tag is `rmiregistry`, which should have been installed along the JRE (Java Runtime Environment) installation. Type the following `ant` command:

```
$ ant rmiregistry
Buildfile: /home/networks/book/build.xml

rmiregistry:

BUILD SUCCESSFUL
Total time: 0 seconds
$
```

The `rmiregistry` should be running as a background process; check it by executing the following `ps` command:

```
$ ps -f
UID          PID  PPID  C  STIME TTY          TIME CMD
networks 23351 23349  0 12:16 pts/0    00:00:00 -bash
networks 23424    1  0 12:49 pts/0    00:00:00 /usr/bin/rmiregistry -J-Djava.rmi.server.useCodebaseOnly=false
networks 23454 23351  0 12:52 pts/0    00:00:00 ps -f
$
```

`Rmiregistry` is running, its PID (Process ID) is 23424.

7. Run server. The server can be launched now. We will use `ant` target “`run_server`”. This task has two direct dependencies, namely “`compile_server`” and “`rmiregistry`”. If we had forgotten to run `rmiregistry` before starting the server, the target's dependencies would have started `rmiregistry` for us, and everything would look the same as in the execution included right below. Launch the server with `ant`'s help, now:

```
$ ant run_server
Buildfile: /home/networks/book/build.xml
```

```
clean_server:
  [delete] Deleting directory /home/networks/book/build/server
  [mkdir] Created dir: /home/networks/book/build/server

compile_server:
  [javac] /home/networks/book/build.xml:58: warning:
'inclideanruntime' was not set, defaulting to
build.sysclasspath=last; set to false for repeatable builds
  [javac] Compiling 3 source files to
/home/networks/book/build/server
  [copy] Copying 1 file to /var/www/html/dsrmipract

rmiregistry:

run_server:
  [java] -----
  [java] - Remote objects instantiated -
  [java] - Remote objects registered (Java RMI registry) -
  [java] - Server running -
  [java] -----
```

The server is running now, in consequence, the two remote methods included in the source code should be available now for remote invocation. We can confidently run a client that sends some remote invocation to the server's exported remote object and expect to receive the return value from the remote method.

8. Compile the client. Generally, testing the client calls for running it in a host other than the server's host ( $H_B$ ). We'll open a session in another host ( $H_A$ ) physically connected to the same LAN as  $H_B$ . This is not a requirement, though: the client and the server can be run within any Internet host as far as they have IP connectivity with each other. Having the client and the server within the same LAN is just a matter of convenience, as it were, given the manifold goals that apply to this example, simplicity being not the least:

The relevant IP addresses are the following:

- $H_A$  : 192.168.1.88
- $H_B$  : 192.168.1.99

We'll open an ssh session in host  $H_A$ , now for us to compile the client.

```
$ ssh your\_login\_name@192.168.1.88
...
```

Download the project files as we did in step 1, then compile the client by calling ant's target "compile\_client":

```
$ ant compile_client
Buildfile: /home/administrator/book/build.xml

clean_client:
  [mkdir] Created dir: /home/administrator/book/build/client
```

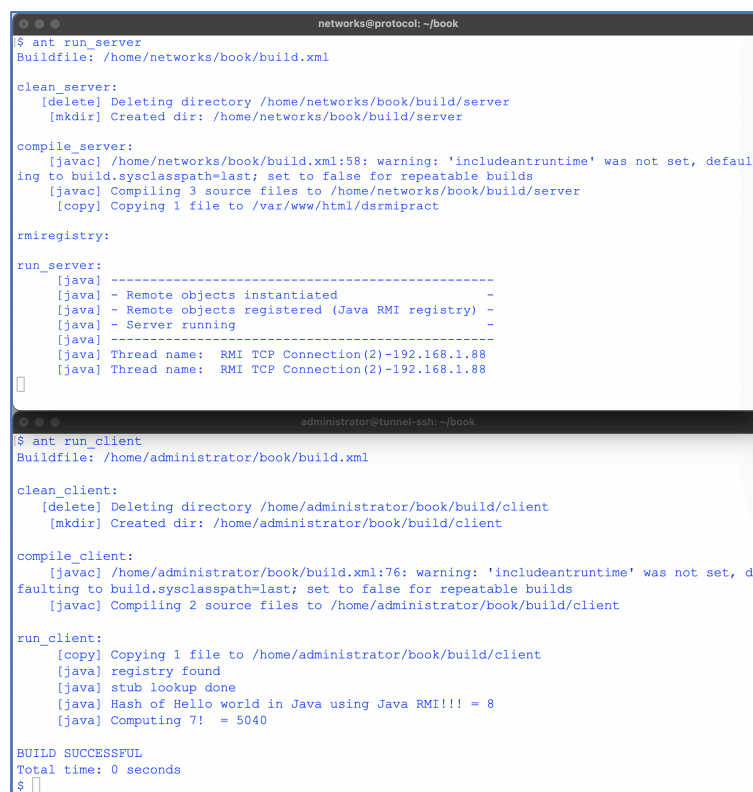


```
compile_client:
  [copy] Copying 1 file to /home/administrator/book/src/client
  [javac] /home/administrator/book/build.xml:76: warning:
'includeantruntime' was not set, defaulting to
build.sysclasspath=last; set to false for repeatable builds
  [javac] Compiling 2 source files to
/home/administrator/book/build/client

BUILD SUCCESSFUL
Total time: 0 seconds
$
```

Target “compile\_client”, first copies the compiled Java interface (File name is SDRRemoteObject.java) so that it gets compiled along with the client’s Java source file. The client uses this interface to cast the remote object’s proxy fetched by accessing rmiregistry. Since compilation was successful, we run the client now which should contact the remote object’s methods by means of the Java RMI runtime.

9. Run the client. Have ant run target “run\_client” by composing the command included in the lower terminal in figure 200. The server program uses the upper terminal window which we launched in step 7.



```
networks@protocol: ~/book
$ ant run_server
Buildfile: /home/networks/book/build.xml

clean_server:
[delete] Deleting directory /home/networks/book/build/server
[mkdir] Created dir: /home/networks/book/build/server

compile_server:
[javac] /home/networks/book/build.xml:58: warning: 'includeantruntime' was not set, default
ing to build.sysclasspath=last; set to false for repeatable builds
[javac] Compiling 3 source files to /home/networks/book/build/server
[copy] Copying 1 file to /var/www/html/dsrmiprac

rmiregistry:

run_server:
[java] -----
[java] - Remote objects instantiated -
[java] - Remote objects registered (Java RMI registry) -
[java] - Server running -
[java] -----
[java] Thread name: RMI TCP Connection(2)-192.168.1.88
[java] Thread name: RMI TCP Connection(2)-192.168.1.88

administrator@tunnel-ssh: ~/book
$ ant run_client
Buildfile: /home/administrator/book/build.xml

clean_client:
[delete] Deleting directory /home/administrator/book/build/client
[mkdir] Created dir: /home/administrator/book/build/client

compile_client:
[javac] /home/administrator/book/build.xml:76: warning: 'includeantruntime' was not set, de
faulting to build.sysclasspath=last; set to false for repeatable builds
[javac] Compiling 2 source files to /home/administrator/book/build/client

run_client:
[copy] Copying 1 file to /home/administrator/book/build/client
[java] registry found
[java] stub lookup done
[java] Hash of Hello world in Java using Java RMI!!! = 8
[java] Computing 7! = 5040

BUILD SUCCESSFUL
Total time: 0 seconds
$
```

Figure 200. Client in lower terminal running against remote object in upper terminal.



This example is comprised of sundry software, some of which was developed from scratch, as well as Java system software, namely the rmiregistry, and finally a Java interface that was to be uploaded to a web server for seamless remote access by application clients. All these software elements were deployed by using an Ant script, flexibly and with ease. In addition to the features that we have showcased here, Ant offers many others. A summary of the most powerful ones is worth which will provide us a glimpse into Ant for future reference.

### Summary of outstanding Ant Concepts and Possibilities

- Cross-platform. The command-line tool that launches the Ant utility is a shell script (Compatible with the POSIX shell standard) that ultimately runs the Ant Java application. In general, ant will run on any platform that runs Java, making it a cross-platform build utility.
- New Ant functionality is attained by adding new Java classes. Extending Java Ant means adding functionality through new Java classes that plug with its current Java class support. The power of Java object-oriented programming is made available for adding new features that might be necessary in specific development circumstances. Since Java Ant is an open source project, all of its current sources are public and, by definition, collaboration in evolving the project is its *raison d'être*.
- Builds are specified in XML files. Ant build files are written in XML by using Ant's specific namespaces. Ant concepts are represented as XML tags. Examples of Ant concepts are *property*, *task*, *target*, and a few others. All of them have a representing XML tag that developers use for expressing which ones are to be launched, with what arguments and in what order so that the intended development semantics is achieved.
- Implementations of Java Ant *task* interfaces allow programmers to specify the execution of actions that lead to a project's specific development stages. Ant is *naturally* extended by extending its classes and interfaces, namely by applying the concepts of OO programming.
- Ant *exec task* allows executing any native operating system command. In the above example, rmiregistry was launched by a Java ant <exec> tag.

### Swift ANT Installation Guide

The Ant software distribution is offered at this URL: <https://ant.apache.org/>. The level of detail to the technical documentation is high and contemplates the habitual sundry of possibilities related with the target operating system release, the underlying hardware platform and user-related install options. If you can afford a mainstream Ant installation and your interest in it stems from your interest to use it for the

practicals included in this book, then you might find it convenient to follow the streamlined installation procedure included below. Follow this *quick* installation outline on the assumption that if problems appear, then you should resort to the *full* instructions provided in the Apache Foundation Ant URL.

We assume that your Linux distribution is one of Debian or Ubuntu. If you use a different one, adapting this installation procedure to it will not be difficult. The output of the `uname` command in our system yields:

```
root@protocol:/home/networks# uname -a
Linux protocol 4.19.0-18-amd64 #1 SMP Debian 4.19.208-1 (2021-09-29) x86_64
GNU/Linux
```

The versions of the Java runtime and the Java compile that we currently use in our system for practicals are the following:

```
root@protocol:/home/networks# java --version
openjdk 11.0.14 2022-01-18
OpenJDK Runtime Environment (build 11.0.14+9-post-Debian-1deb10u1)
OpenJDK 64-Bit Server VM (build 11.0.14+9-post-Debian-1deb10u1, mixed mode,
sharing)
root@protocol:/home/networks# javac --version
javac 11.0.14
```

The Ant release we will install is 1.10.x which fits the requirements for Java 8 and newer versions; an overall installation procedure follows:

1. Download the zip file that contains the Ant distribution

<https://dlcdn.apache.org/ant/binaries/apache-ant-1.10.12-bin.zip>

2. Download the digital keys for verifying the downloaded zip file

<https://downloads.apache.org/ant/KEYS>

3. Download the asc file that carries the pgp key (Pretty Good Privacy)

<https://downloads.apache.org/ant/binaries/apache-ant-1.10.12-bin.zip.asc>

4. Install the pgp (Pretty Good Privacy) distribution:

```
Switch users to super-user
root@protocol:/home/networks# su
```

```
Update software packages
root@protocol:/home/networks# apt update
```

```
Upgrade the installed software packages themselves
root@protocol:/home/networks# apt upgrade
```

```
Install the gpg software package
root@protocol:/home/networks# apt install gpg
```

Check that the zip file, the asc file and the KEYS file are in the present directory

```
root@protocol:/home/networks# ls
apache-ant-1.10.12-bin.zip  apache-ant-1.10.12-bin.zip.asc  KEYS
```

5. Check the ant distribution's zip file digital signature:

```
root@protocol:/home/networks# gpg --verify apache-ant-1.10.12-bin.zip.asc
gpg: assuming signed data in 'apache-ant-1.10.12-bin.zip'
gpg: Signature made Wed 13 Oct 2021 06:50:47 AM CEST
gpg: using RSA key 8DA70C00DF7AF1B0D2F9DC74DDBC1270A29D081
gpg: Good signature from "jaikiran@apache <jaikiran@apache.org>"
[unknown]
gpg: WARNING: This key is not certified with a trusted signature!
gpg:          There is no indication that the signature belongs to
the owner.
Primary key fingerprint: 8DA7 0C00 DF7A F1B0 D2F9 DC74 DDBC C127 0A29 D081
```

The signature is good, however, the owner's identity could not be verified, so our trust will be based only on the merit that the software was downloaded from the Apache Foundation's official site. Nevertheless, we should triple check that, if our installation were for a productivity environment.

6. The manual pages to the Ant distribution can be found within the downloaded distribution, under the *manual* directory. The manual can also be accessed from the web at this link:

<http://ant.apache.org/manual/index.html>

7. Unzip the distribution files:

```
root@protocol:/home/networks# unzip apache-ant-1.10.12-bin.zip
root@protocol:/home/networks# cd apache-ant-1.10.12-bin
root@protocol:/home/networks# ls
bin  etc  INSTALL  LICENSE  patch.xml
CONTRIBUTORS  fetch.xml  KEYS  manual  README
contributors.xml  get-m2.xml  lib  NOTICE  WHATSNEW
```

8. Change directories to bin directory and execute Ant from there for checking that it is installed and for checking its version, also:

```
root@protocol:/home/networks# cd bin
root@protocol:/home/networks/bin# ./ant -version
Apache Ant(TM) version 1.10.12 compiled on October 13 2021
```

9. Copy the files that were uncompressed from the zip file to /usr/local/ant and export the ANT\_HOME variable to the shell's environment; finally, for convenience, have it added to your PATH:

```
root@protocol:/home/networks/bin# cd ..
root@protocol:/home/networks# mkdir /usr/local/ant
root@protocol:/home/networks# cp -r * /usr/local/ant
root@protocol:/home/networks# export ANT_HOME=/usr/local/ant
root@protocol:/home/networks# export PATH=$PATH:$ANT_HOME
```

*Include the previous exports in your .bashrc startup file so that both are executed every time you request a new terminal*

*Logout now and log back in to check that the intended ant configuration is available; check by calling ant from a new terminal as in Fig. 300.*

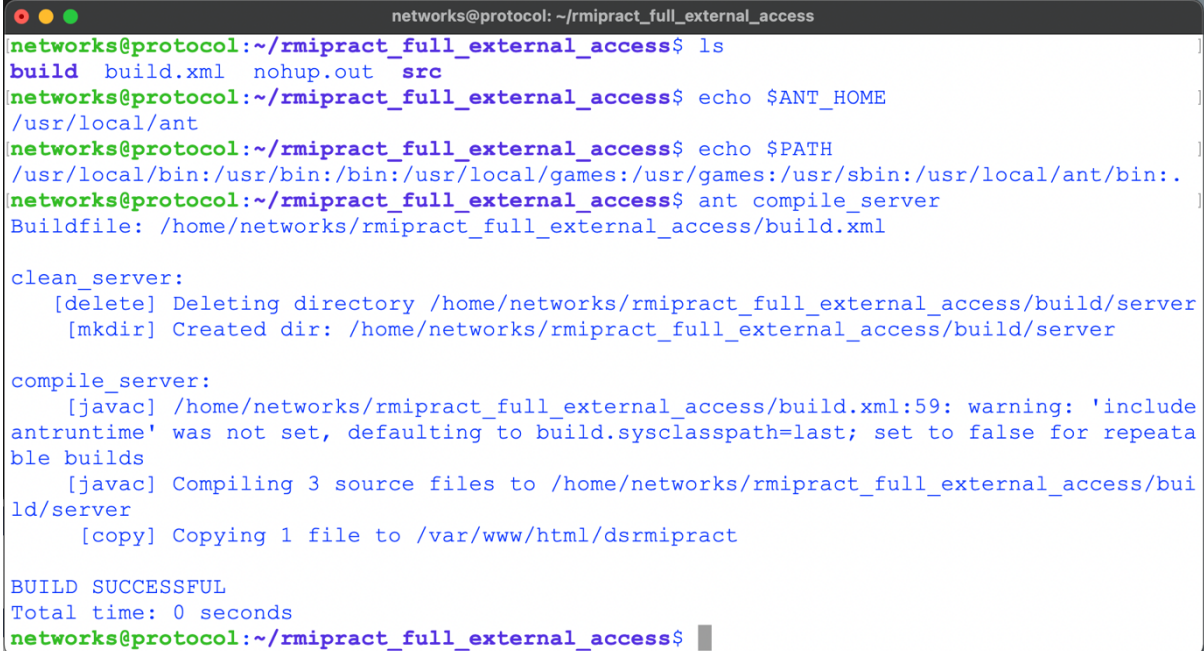
A terminal window titled 'networks@protocol: ~/rmipract\_full\_external\_access'. The user runs 'ls' showing 'build build.xml nohup.out src'. Then 'echo \$ANT\_HOME' returns '/usr/local/ant'. Then 'echo \$PATH' returns '/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games:/usr/sbin:/usr/local/ant/bin:.'. Then 'ant compile\_server' runs, showing 'Buildfile: /home/networks/rmipract\_full\_external\_access/build.xml', 'clean\_server:' with directory deletion and creation, and 'compile\_server:' with javac warnings and compilation of 3 source files to /var/www/html/dsrmiprac. It ends with 'BUILD SUCCESSFUL' and 'Total time: 0 seconds'.

Figure 300. Ant checks after installation