User Guide for CAN-GRID

Michael Marsh

September 24, 2007

Contents

1 Compiling — Turning the source code into something useful **2**
1.1 The short version, if you're feeling lucky. 2
1.2 The longer version, if you're not sure what's going on. 2

2 Preparing to use the system **5**
2.1 Why do we have an authentication system? 5
2.2 How does the authentication system work? 5
2.3 Cutting to the chase: How do we use this? 5

3 Running a peer node **7**
3.1 Starting a peer. 7
3.2 Forming the grid. 8
3.3 Peer configuration files 8

4 Running a user client **10**
4.1 Starting a client 10
4.2 Job specifications. 10
4.3 Selecting a peer. 11
4.4 The message pane. 11
4.5 Client configuration files 12

5 Job description files **13**
5.1 Resource requirements. 14
5.2 Files. 14
5.3 Invocation. 14
5.4 A complete XML file. 15
5.5 Generating XML files. 16
5.5.1 Setting Resource Requirements 16
5.5.2 Input Files 16
5.5.3 Invocation 18
5.5.4 Generating the Output File 19

6 Command-line arguments **21**
6.1 peer 21

Chapter 1

Compiling — Turning the source code into something useful

1.1 The short version, if you're feeling lucky.

The short version is to run the following sequence of commands, where "$" is the shell prompt:

```
$ ./configure
$ make
$ make client_gtk
```

and optionally (if you want the source code documentation):

```
$ make doc
```

1.2 The longer version, if you're not sure what's going on.

The longer version is that autoconf is being used to make the source code more portable from one platform to another. The command

```
$ ./configure
```

runs the autoconf-generated script that sets up the build system and determines a few details specific to a given computer. This script can take a number of options, which can be displayed by running

```
$ ./configure --help
```

Most of this "help" output will not be useful, so here we reproduce the more interesting items:

```
--enable-debug           If enabled, compile with debug flags. Otherwise,
                         compile with optimization.
--with-ssldir=<path>     Look for OpenSSL libs and includes in <path>.
--with-ssllibdir=<path>  Look for OpenSSL libraries in <path>.
--with-sslincdir=<path>  Look for OpenSSL includes in <path>/openssl.
--with-readlinedir=<path>
                         Look for readline libs and includes in <path>.
--with-readlinelibdir=<path>
                         Look for readline libraries in <path>.
--with-readlineincdir=<path>
                         Look for readline includes in <path>/readline.
```

```
  --with-pkgconfig=<prog> Use <prog> for pkg-config.

Some influential environment variables:
  READLINE    Link-time flags needed to use readline.
  DOXYGEN     Name of the Doxygen executable.
```

If you're lucky, you won't need to specify any of these. If you plan to make changes to the code, you might want to use `--enable-debug`. However, if you plan to make changes to the code, you probably didn't need to be told this. You can run configure multiple times, so the best thing to do is to try running it without any options, and add the options you need later if you encounter problems.

Most POSIX-type operating systems now come with OpenSSL more-or-less standard. However, yours might not, or it might have been installed somewhere that the compiler can't find it. In this case, you'd want to use the `--with-ssldir` option. For example, if your OpenSSL libraries are installed in `/opt/ssl/lib` and headers in `/opt/ssl/include/openssl`, then you'd run:

```
$ ./configure --with-ssldir=/opt/ssl
```

The library and header directories can be specified independently with:

```
$ ./configure --with-ssllibdir=/opt/ssl/lib --with-sslincdir=/opt/ssl/include
```

The configure script might fail for other reasons. The most likely reason for failure is a missing package. The following packages are needed to successfully compile and link the programs:

OpenSSL used for authentication

pthreads the POSIX threading library

readline used by the menu-driven client

Depending on the version of readline that you have, and how it was built, you might also need the **curses** package.

Another optional package is **Doxygen**. This generates documentation from special comments within the source code. If you don't have it installed, don't worry. The `make doc` command will not work, though. If you have Doxygen installed, but configure can't find it (or finds a version you don't want to use), you can tell it explicitly to use a particular version. For example, if the version of Doxygen that you want to use is `/opt/doxygen-1.4/doxygen`, you could run:

```
$ ./configure DOXYGEN=/opt/doxygen-1.4/doxygen
```

A tip for if you need to run configure multiple times: After each time it's run, configure creates a file `config.log` that includes the command that was actually run. As is the case here, "$" represents the shell prompt. This allows you to cut and paste the old command line, and add any new options that are required.

Once configure has run successfully, it's time to build the executables. This is done by running:

```
$ make
```

You should see no warning or error messages from the compiler or linker. The output to your screen should look something like:

```
Updating dependencies...
...done
gcc -g  -Wall -Werror -ansi -pedantic -c auth.c
gcc -g  -Wall -Werror -ansi -pedantic -c callback.c
gcc -g  -Wall -Werror -ansi -pedantic -c can.c
gcc -g  -Wall -Werror -ansi -pedantic -c job.c
gcc -g  -Wall -Werror -ansi -pedantic -c neighbors.c
gcc -g  -Wall -Werror -ansi -pedantic -c network.c
```

```
gcc -g  -Wall -Werror -ansi -pedantic -c new_del.c
gcc -g  -Wall -Werror -ansi -pedantic -c queue.c
gcc -g  -Wall -Werror -ansi -pedantic -c self.c
gcc -g  -Wall -Werror -ansi -pedantic -c serialize.c
gcc -g  -Wall -Werror -ansi -pedantic -c unserialize.c
gcc -g  -Wall -Werror -ansi -pedantic -c util.c
gcc -g  -Wall -Werror -ansi -pedantic -lm -lpthread -lcrypto auth.o callback.o
can.o job.o neighbors.o network.o new_del.o queue.o self.o serialize.o unserial
ize.o util.o -o peer peer.c
gcc -g  -Wall -Werror -ansi -pedantic -lm -lpthread -lcrypto -lreadline -lcurse
s client_ops.o auth.o callback.o can.o job.o neighbors.o network.o new_del.o qu
eue.o self.o serialize.o unserialize.o util.o -o client client.c
gcc -g  -Wall -Werror -ansi -pedantic -lm -lpthread -lcrypto client_ops.o auth.
o callback.o can.o job.o neighbors.o network.o new_del.o queue.o self.o seriali
ze.o unserialize.o util.o -o client_join client_join.c
gcc -g  -Wall -Werror -ansi -pedantic -lm -lpthread -lcrypto client_ops.o auth.
o callback.o can.o job.o neighbors.o network.o new_del.o queue.o self.o seriali
ze.o unserialize.o util.o -o client_leave client_leave.c
```

For a peer installation, this is sufficient (you'll want `client_join`, and probably `client_leave`). For a system on which you'll want to run the graphical client, run

```
$ make client_gtk
```

This, of course, requires that **Gtk** (2.0 or above) is installed.

If you have Doxygen installed, and you want to create the source-code documentation, you can now run:

```
$ make doc
```

This will produce a directory full of HTML-format documentation. The main page will be

```
doc/html/index.html
```

under the directory with the source code.

Chapter 2

Preparing to use the system

We have tried to make the system as easy to use as possible, but there is a certain amount of inherent complexity. The biggest source of complexity for the user is the authentication system.

2.1 Why do we have an authentication system?

We make some fairly lax assumptions regarding security. In particular, we're more worried about admitting machines and people into the system than whether they'll misbehave later. That means we want to make sure a computer is recognized as an acceptable participant (that is, authenticate it) when it tries to join the system. We want to authenticate a person as an acceptable user when he or she asks the system to run a compute job.

2.2 How does the authentication system work?

We use digital signatures to authenticate principals (computers or users). There is one principal that is recognized as the "owner" of the particular grid instance. That owner has a digital certificate, which is signed by its own corresponding private key. The certificate contains a public key that other principals can use to verify the owner's signature.

The other principals choose pairs of public and private keys. They then ask the owner to issue certificates for their public keys. The owner can generate two types of certificates: contributor certificates and user certificates. The former are used to identify a computer as an acceptable contributor of computing resources. The latter are used to identify a person as an acceptable user of those resources.

When a principal needs to digitally sign a message, it uses its private key, and sends the signature and public key certificate along with the message. Other principals can then verify that the certificate is valid, and that it can confirm the signature sent with the message.

2.3 Cutting to the chase: How do we use this?

We have provided a set of scripts to do most of the work for you. These all require the OpenSSL package. There is a `keys` directory that comes with the source code. **DO NOT USE THESE KEYS!** They're for testing purposes only! The private keys do not have passwords set, which means they're almost completely insecure. We're not going to tell you how to create password-less private key files, because you shouldn't be using them.

IMPORTANT NOTE: The files that we'll generate with the scripts in this section are specially formatted. Even though most of them will be printable ASCII, their relevant content is actually binary data. OpenSSL (which is the package we're using) calls this *ASCII armoring*. In order for OpenSSL to properly read an ASCII-armored file, the plain-text lines at the beginning and end **must** be present. Files should be transferred in their entirety. When mailing, the safest thing to do is to include the files as attachments, rather than pasting them inline.

If you're going to be the owner of a grid, you will need to run:

```
$ create_owner_certificate
```

This will generate a key pair and a certificate. When creating a certificate, you'll see the following prompts:

```
Country Name (2 letter code) [GB]:
State or Province Name (full name) [Berkshire]:
Locality Name (eg, city) [Newbury]:
Organization Name (eg, company) [My Company Ltd]:
Organizational Unit Name (eg, section) []:
Common Name (eg, your name or your server's hostname) []:
Email Address []:
```

Set these as appropriate. The private key will be written to `owner_privkey.pem`, and the certificate to `owner_cert.pem`. `owner_cert.pem` should be given to everyone who will be running compute nodes or submitting jobs.

If you're going to be running compute nodes for the grid or want to be able to submit compute jobs, you will need to run:

```
$ create_key_pair
```

You'll see the prompts listed above for `create_owner_certificate`, as well as:

```
A challenge password []:
An optional company name []:
```

It is fine (preferred, even) to leave these blank. At this point, you'll have a file `privkey.pem` containing your private key (don't forget the password!), and a file `pubkey.req`. The latter contains your new public key and all of the information for which you were prompted, and is formatted as a request for a certificate to be generated.

Now you'll need to send `pubkey.req` to the grid owner. The grid owner will then run either

```
$ create_contributor_certificate pubkey.req owner_cert.pem owner_privkey.pem
```

or

```
$ create_user_certificate pubkey.req owner_cert.pem owner_privkey.pem
```

The resulting certificate will be called either `contrib_cert.pem` or `user_cert.pem`, depending on which type of certificate was generated. This certificate file should be sent back to the person who sent the request `pubkey.pem`. It is possible that both will need to be generated.

Note that any of the file names can be changed after they've been generated. The commands will have to be changed accordingly, of course.

Chapter 3

Running a peer node

3.1 Starting a peer.

In order to run a peer node in the computational grid, you need the following:

- the `peer` exectuable, compiled and linked for your platform
- a contributor certificate file, which we will refer to as `contrib_cert.pem`, but which you can rename anything you like
- a private key file, containing the private key corresponding to the public key in `contrib_cert.pem`. We will refer to this file as `contrib_privkey.pem`.
- a certificate file for the grid owner, which we will refer to as `owner_cert.pem`

Creating the certificate and private key files is described in 2.3.

The peer is typically configured through a file. This file tells the peer where to find the certificates and private key, on which ports to listen, and what computational resources to offer. See `examples/peer_example.conf` for a sample configuration file (with comments). We will discuss the various configuration options in section 3.3.

The peer listens on two ports. One of these is the *client port*, and the other is the *peer port*. While the grid is a peer-to-peer system, individual peers retain some server-like properties, in that some operations are performed by client processes. For example, bootstrapping into the network (covered in the next section) is initiated by a client command. Most significantly, jobs are submitted by clients. In effect, clients act as the peers' user interfaces. The nominal default client port is **6767**.

The peer port, by contrast, is for messages involving the peer-to-peer operations of the grid. The nominal default peer port is **6771**. The same message might be interpreted differently depending on the port to which it's sent.

If the executable resides in the directory `/opt/grid`, and the configuration file is `/etc/grid.conf`, you would start the grid node with the command:

```
$ /opt/grid/peer -f /etc/grid.conf
```

Optionally, you might share a configuration file among a number of nodes, each with slightly different configurations. The configuration file can accommodate this with named sections. If you use the short hostname as the section, you would start the node on `a.foo.edu` with the command:

```
$ /opt/grid/peer -f /etc/grid.conf -s a
```

The peer can also be configured on the command line. If both types of configuration are specified, the command-line arguments will override the values in the configuration file. The command-line arguments are discussed in section 6.1

3.2 Forming the grid.

The CAN network does not handle merges of existing smaller networks well, so a small amount of care is required to construct the grid.

A grid of a million nodes begins with a single host. Each peer, when first started, assumes that its zone covers the entire grid space. Someone, probably the grid owner, will start the first node, which we'll call *alpha*. The next node, *beta*, will have to "bootstrap" through alpha. Subsequent nodes can bootstrap into the grid through *any* existing node, so the next node, *gamma*, can bootstrap through either alpha or beta.

Since a grid is likely to span institutions, or at least smaller administrative units, we illustrate the following deployment procedure:

1. Institution A, which owns the grid, sets up its nodes.
2. Institution B then connects one node, which we'll call B0, to the grid through any node at institution A.
3. Subsequent nodes at institution B bootstrap through B0.

By bootstrapping the nodes after B0 through B0, we guarantee that the first network connection in the bootstrapping process is over a local link. In fact, if all of institution B's nodes have similar resource capabilities, it is likely that all of its nodes will occupy the same region of the CAN space, and most of the bootstrapping messages will be sent over local (and hence faster) links.

Nodes bootstrap themselves in response to a command from a client, which must include the address of an existing member of the grid. A simple client, called `client_join`, provides this functionality. Let's say that we want to bootstrap `zero.b.edu` into the system, using an existing node `three.a.edu`. Both peers will listen for client connections on port 6767, and peer connections on 6771. We would then issue the command:

```
$ /opt/grid/client_join -h zero.b.edu -p 6767 -H three.a.edu -P 6771
```

That is, we connect to the peer at host `zero.b.edu` on port 6767 and send it a join command with host `three.a.edu` and port 6771 as the bootstrap node. `client_join` may be run from any host that can connect to `zero.b.edu`. See `examples/bootstrap_cluster.sh` for a sample script that bootstraps a cluster of nodes into a system.

3.3 Peer configuration files

All configuration options are specified in the format:

```
key = value
```

We now describe the various configuration keys recognized by the peer.

`host` The name (or IP address) of the host on which the peer runs.

`private_key` The file containing the peer's private key.

`certificate` The file containing the peer's public key certificate, signed by the grid owner.

`owner_certificate` The file containing the grid owner's public key certificate.

`client_port` The port on which the peer listens for connections from clients.

`peer_port` The port on which the peer listens for connections from other peers.

`CPU` The advertised CPU speed of the host, in MHz.

`memory` The advertised RAM provided by the host, in MB.

`storage` The advertised disk space provided by the host, in XXX.

`verbosity` The amount of detail the peer should output. The following details the types of output for the different verbosity levels:

1. basic start-time peer configuration; changes to the peer's zone in the CAN space
2. error reporting
3. peer joins, leaves, and failure takeovers; job insertion, matchmaking, and running; signing and verifying messages
4. neighborhood changes; heartbeat timestamps
5. heartbeating for CAN maintenance and job maintenance
6. message handling; configuration debugging; finer-grained CAN debugging
7. greater detail on message signing and verification
8. acquiring and releasing mutex locks

`jail` If 0, do not run user jobs in a `chroot` jail. If greater than 0, the `suexec` program is used to jail user jobs, isolating them from the filesystem.

Chapter 4

Running a user client

4.1 Starting a client

For the client, you need the following:

- the `client_gtk` executable, compiled and linked for your platform
- a certificate file for a user, which we will refer to as `user_cert.pem`
- a private key file, containing the private key corresponding to the public key in `user_cert.pem`, which we will refer to as `user_privkey.pem`
- a certificate file for the grid owner, which we will refer to as `owner_cert.pem`
- the hostname where a peer is running (`zero.b.edu`)
- the port on which the peer listens for clients (6767)

As with the peer, the client is configured through a file. The file tells the client where to find certificate and key files. Optionally, it also contains a default peer. It may also contain port information for responses from the system.

The client is started with the command:

```
$ /opt/grid/client_gtk -f client.conf
```

See `examples/client_example.conf` for a sample configuration file (with comments). We discuss the configuration options in section 4.5.

Users must have some way of retrieving results through the client. How this is done depends on in which of two modes the client is running. In the "standard" mode, the client is always connected to the network, and listens for results on an open port. In addition to receiving job results, the client will also periodically receive and respond to messages from peers at which the client's jobs are running or scheduled.

Sometimes the client will not be able to run in standard mode, either because it cannot be left running continually, or because a firewall or network address translator (NAT) sits between the client's host and the network. In this case, the client should run in "proxied" mode. In this mode, the client requests that a peer act as its proxy. This means the peer will exchange messages with run nodes on the client's behalf, and will cache job results for the client. Selecting between standard and proxied mode is covered in section 4.5.

4.2 Job specifications.

When started, the client opens a window with a job specification area, a peer selection area, and a message pane. Jobs are specified as XML files, and each file may contain multiple jobs. Typically, you will use the provided java

application to generate the XML. Both the structure of the XML files and the utility to generate the files are discussed in chapter 5.

You can type (or paste) the name of a job description file into the entry area, or you can use the "Browse" button to open the file picker (Figure 4.1). The file picker allows you to navigate through the directory structure on your computer to find the appropriate jobfile. The "Batch Submit" button will then send the job or jobs described in the file to the currently selected peer.

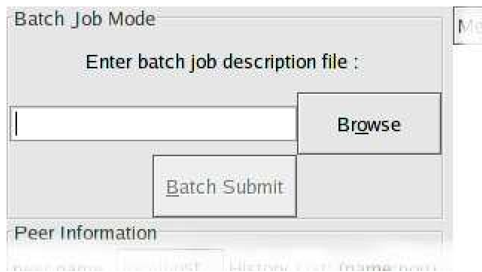

Figure 4.1: Job submission frame.

If the client is running in proxied mode, there will also be a "Pull Results" button. This polls the currently selected peer (see Section 4.3) for job results. If that peer is caching any results for the client, it will return them and clear the results from its cache. If you have submitted jobs through multiple peers, then you will need to poll each of them for any cached results. In the non-proxied mode, results will be returned directly to the client.

4.3 Selecting a peer.

The peer information frame (Figure 4.2) holds the currently selected peer name and port number. To change this, click on the "Change" button and enter the new information in the text entry areas. You can click "Cancel" to revert to the previous configuration or "Set" to update the selection.

When a new peer is set, it is also added to the history list drop-down. Selecting a peer from this list and clicking on "<<Set" will copy the peer information into the current configuration.

You may change peers as often as you like, but if you've requested proxying, then you must poll the peers through which you've submitted jobs for any cached results. In general, when proxying it is advisable to stick with a single peer.

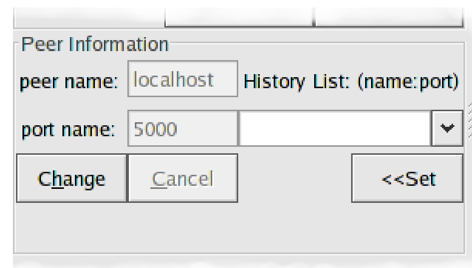

Figure 4.2: Peer information frame.

4.4 The message pane.

Messages are displayed on the right side of the client window (Figure 4.3. Each message appears as a color-coded timestamp (blue for informational messages, red for errors) with a triangular widget to its left. Clicking on the widget expands the entry to show the text of the message. Messages can be repeatedly expanded and collapsed, and scrollbars appear as needed.

There is also a "Save" button that allows you to capture the entire message log into a file.

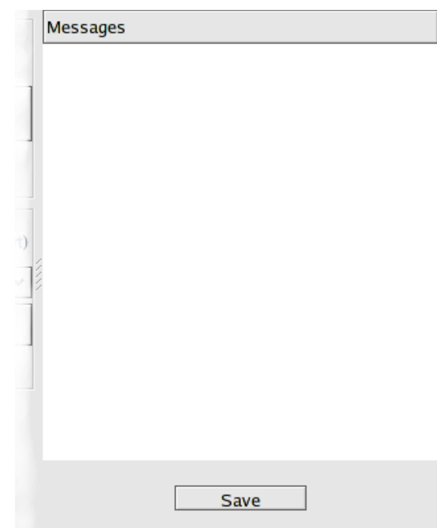

Figure 4.3: Message frame.

4.5 Client configuration files

As with the peer configuration files (Section 3.3), options are specified as:

```
key = value
```

The keys recognized by the client are:

private_key The file containing the user's private key.

certificate The file containing the users's public key certificate, signed by the grid owner.

owner_certificate The file containing the grid owner's public key certificate.

peer_name [optional] The hostname (or IP address) of a default peer. Additional peers may be specified through the interface. `peer_port` must also be specified.

peer_port [optional] The port on which the default peer listens for clients. `peer_name` must also be specified.

hostname [standard mode] The hostname (or IP address) for the machine on which the client is running.

port [standard mode] The port on which to listen for responses and job heartbeat messages.

In standard mode, both `hostname` and `port` must be given. If only one is provided, the client will exit with an error. If neither is provided, the client will run in proxied mode. See Section 4.1 for details. The hostname must be given to the client explicitly because there is no standard method for determining it. A simple way to get around this on a POSIX system is to set

```
hostname = $ENV::HOSTNAME
```

in the configuration file, and start the client from a shell script with an invocation like

```
/usr/bin/env HOSTNAME=`hostname` /opt/grid/client_gtk -f client.conf
```

Chapter 5

Job description files

The bulk of this section is intended as a reference for the XML job description file format. If you only care about the java program to generate these files for you, and don't plan to ever look at or modify an existing XML file, you can skip ahead to section 5.5.

The general structure of a file is:

```
<jobset>

  <job>
     ...job specification...
  </job>

  <job>
     ...job specification...
  </job>

  ...

</jobset>
```

Each job has the following structure:

```
  <job>

    <requirements>
      ...resource requirements...
    </requirements>

    <files>
      ...files to download/link...
    </files>

    <invocation>
      ...command to run...
    </invocation>

  </job>
```

5.1 Resource requirements.

The requirements section specifies the minimum resources for a node that will run the job:

```
<requirements>
  <cpu>500</cpu>
  <mem>128</mem>
  <disk>50</disk>
</requirements>
```

The above specifies that the job requires a 500MHz processor or better, at least 128MB of RAM, and at least 50XXX of disk space. This includes the space needed for the executable and all input and output files.

5.2 Files.

The files section lists the files that will be needed by the job. Several file types are recognized, and files may optionally have labels. Labels will be discussed in greater detail in the context of the invocation section (5.3).

A file section might look like:

```
<files>
  <file label="exe" type="local">/usr/bin/wc</file>
  <file label="input" type="url">http://www.google.com</file>
</files>
```

A "local" file is one that should be present on the **peer** (*not* the client). In this case, it's the standard Unix "word-count" utility. A URL will be downloaded by the run node. The other recognized file types, "reference" and "inline", are not used in the files section.

The file section is optional, though most jobs will have one.

5.3 Invocation.

This section tells the run node what to run, and hence is mandatory for any job. Its structure is:

```
<invocation>
  <binary type="...">...</binary>
  <args>
    <arg type="...">...</arg>
    ...
  </args>
</invocation>
```

The "binary" tag is a file specification, as above. Only "local" and "reference" are valid file types, and the "label" attribute is ignored. If the binary type is "reference", then the file name must be a label defined in the "files" section. For example:

```
<binary type="reference">exe</binary>
```

An "arg" tag specifies a command-line argument to be passed to the binary. Arguments are passed in the order specified in the file. An argument may have type "literal" or "reference". As with "binary", a reference should specify a file label, such as:

```
<arg type="reference">input</arg>
```

A literal argument is just a string that's passed as-is to the binary:

```
<arg type="literal">-l</arg>
```

5.4 A complete XML file.

Putting the above together, we might have the following XML file containing a single job:

```
<jobset>

  <job>
    <requirements>
      <cpu>500</cpu>
      <mem>128</mem>
      <disk>50</disk>
    </requirements>
    <files>
      <file label="exe" type="local">/usr/bin/wc</file>
      <file label="input" type="url">http://www.google.com</file>
    </files>
    <invocation>
      <binary type="reference">exe</binary>
      <args>
        <arg type="literal">-l</arg>
        <arg type="reference">input</arg>
      </args>
    </invocation>
  </job>

</jobset>
```

This job would download the Google homepage (say to a file named “input”), and would run the following command:

```
/usr/bin/wc -l input
```

5.5 Generating XML files.

The easy way to generate job description files is using the `java/generate_jobfile` program. This is a script that launches a java application contained in the file `java/jobgen.jar`. If this file was not present in the copy of the code that you obtained, then you can build it by going into the `java` subdirectory and running `make`. The java bytecode can be run on any platform with java version 1.5 or later. The initial screen is shown in Figure 5.1 At any time, you can hit the "Clear" button to reset all fields to this initial configuration, or "Quit" to exit without generating a file.

We will examine each of the areas of this interface in detail in the following subsections. We'll start at the top of the screen and work our way downwards, which represents the typical order in which the job components will be specified.

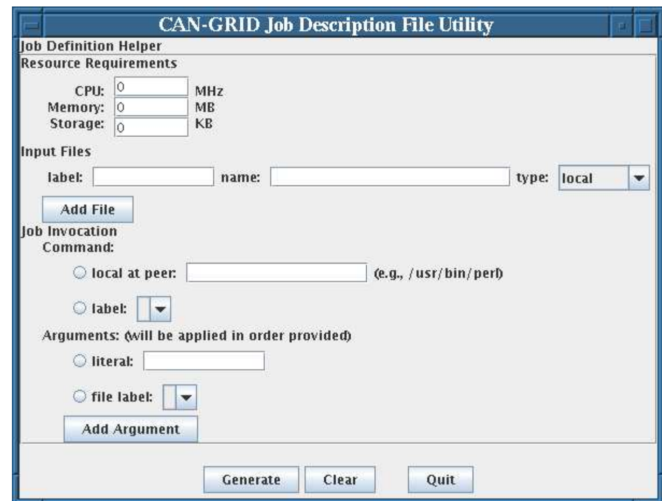

Figure 5.1: Jobfile generator.

5.5.1 Setting Resource Requirements

By default, no requirements are placed on a job. If you want to set a minimum requirement, edit the appropriate text entry field. For example, Figure 5.2 demonstrates setting a minimum of 100MB of RAM. All jobs generated will have the same resource requirements, so there is nothing else to do to specify these requirements.

Obviously, the fewer (or less restrictive) requirements you specify, the larger the pool of potential peers to run your job. Since many jobs will have some sort of disk usage requirement for input, output, or temporary files, the *Storage* resource is likely to be set.

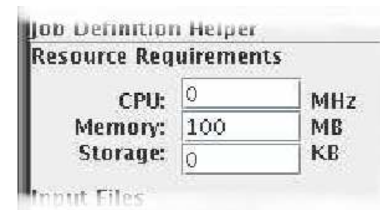

Figure 5.2: Setting resource requirements.

5.5.2 Input Files

Figure 5.3: Specifying a file label and location.

Files specified here will be downloaded or copied as necessary. As an example, we want to specify a file to which we'll apply the label "exe," and which can be downloaded via HTTP (Figure 5.3). The URL that we provide (`http://jobs.my.domain/job1`) will be retrieved with the `wget` command and saved to the file name `exe`. The available file types are:

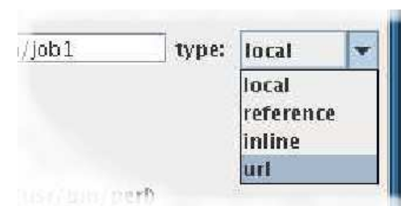

Figure 5.4: Setting the file type.

local A file that should exist on the *peer* that will run the job.

reference An existing file label, which does not make sense in this context.

inline A file where the "name" is the complete file content.

url A uniform resource locator, such as an `http:` or `ftp:` address. The file will be downloaded by the peer when the job is selected to run.

The default file type is “local,” which is not what we want. We want to specify this file as having type “url,” so we select it from the drop-down menu, as shown in Figure 5.4.

Now that we’ve specified this file completely, we need to add it to the file list by pressing the “Add File” button, as shown in Figure 5.5. The file will now appear in the list of defined files (Figure 5.6). If the file has a label, it’s possible to add additional files with the same label by clicking on the small “+” button next to the file name. This button will expand the interface next to the label to provide an additional text box for a new file name, another menu to select the file type, and button to add the new file specifically to that label. The files that share a label need not be the same type. The “+” button also changes into a “-” button, which will collapse this addition to the interface.

Figure 5.5: Adding a file.

Figure 5.6: Expanding the interface for an existing label.

Entering the file information and clicking the “Add File” button in the label-specific area adds the new file name under the existing label, as shown in Figure 5.7. With two files sharing a label, the program will generate XML with twice the jobs: half with the first value of “exe” and half with the other. If we add a third file, then we’d triple the number of jobs.

Figure 5.7: Adding an additional file to an existing label.

We can add another file, with the label “data,” using the same procedure. As before, we can specify multiple files with the same label (Figure 5.8).

At present, we’ve (partially) defined four jobs, corresponding to the combinations of `job1` and `job2` for the label “exe” and `data` and `more_data` for the label “data.”

It is important to note that, unlike for resource requirements, *every* file must be added with an “Add File” button. It is also possible to add unlabelled files, but these cannot take multiple values as demonstrated above.

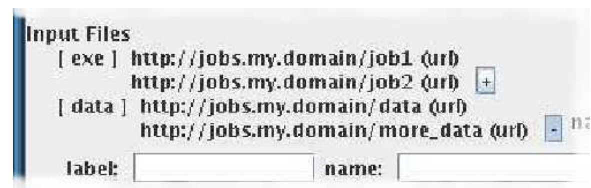

Figure 5.8: Another label with two files.

5.5.3 Invocation

This part of the interface lets you specify what job to call, and how to call it. We'll begin with the command to invoke. There are two options: a file that already exists on the *peer* where the job will run, and the label of a file specified in the "Input Files" section.

A local file might be the perl interpreter, a java virtual machine, or any other standard tool. More likely, you'll want to select a file that you have labelled (which may itself be a local file). Whichever option you choose, click on the appropriate radio button. For a file label, select the appropriate file from the drop-down menu next to the label option (Figure 5.9). This menu is populated for you with all of the labels that have been defined. There is only one command specified for a jobset, so there is no "Add Command" button to press.

Figure 5.9: Selecting a file label as the executable.

Next we look at specifying arguments to pass to the job on the command line. Arguments are passed to the specified command in the order they are added. As with commands, there are two options for an argument: literal or file label. A literal argument (see Figure 5.10) is just a text string that will be included verbatim on the command line. Since we may have several arguments, you will have to press the "Add Argument" button in order for the argument to be included.

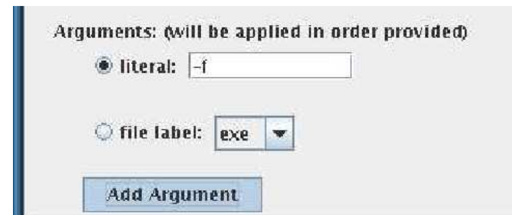

Figure 5.10: A string-literal argument.

Once added, a literal argument may be given multiple values, just as with a file label. This is shown in Figure 5.11. If multiple versions of an argument are given, then we once again multiply the number of jobs, with the different values used in that command-line position. This is a straightforward way of constructing a series of jobs to perform, for instance, a parameter sweep. Specifying multiple values for several parameters results in an expansion to all parameter value combinations.

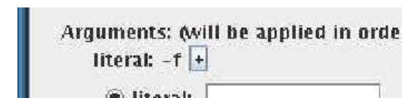

Figure 5.11: Alternate values for a string-literal argument.

File labels can also be added as arguments (Figure 5.12), though these cannot have different values. If you want multiple values for a file label, this should be specified under "Input Files."

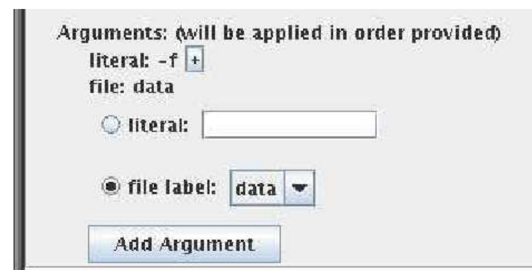

Figure 5.12: A file label as a command-line argument.

5.5.4 Generating the Output File

At this point, we are ready to generate the XML file. We do this by clicking on the “Generate” button (Figure 5.13). The resulting XML file is:

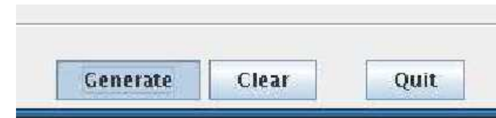

Figure 5.13: Generating the file.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<jobset>

  <job>
    <requirements>
      <cpu>0</cpu>
      <mem>100</mem>
      <disk>0</disk>
    </requirements>
    <files>
      <file label="exe" type="url">http://jobs.my.domain/job1</file>
      <file label="data" type="url">http://jobs.my.domain/data</file>
    </files>
    <invocation>
      <binary type="reference">exe</binary>
      <args>
        <arg type="literal">-f</arg>
        <arg type="file">data</arg>
      </args>
    </invocation>
  </job>

  <job>
    <requirements>
      <cpu>0</cpu>
      <mem>100</mem>
      <disk>0</disk>
    </requirements>
    <files>
      <file label="exe" type="url">http://jobs.my.domain/job2</file>
      <file label="data" type="url">http://jobs.my.domain/data</file>
    </files>
    <invocation>
      <binary type="reference">exe</binary>
      <args>
        <arg type="literal">-f</arg>
        <arg type="file">data</arg>
      </args>
    </invocation>
  </job>


  <job>
    <requirements>
      <cpu>0</cpu>
      <mem>100</mem>
      <disk>0</disk>
```

```
    </requirements>
    <files>
      <file label="exe" type="url">http://jobs.my.domain/job1</file>
      <file label="data" type="url">http://jobs.my.domain/more_data</file>
    </files>
    <invocation>
      <binary type="reference">exe</binary>
      <args>
        <arg type="literal">-f</arg>
        <arg type="file">data</arg>
      </args>
    </invocation>
  </job>

  <job>
    <requirements>
      <cpu>0</cpu>
      <mem>100</mem>
      <disk>0</disk>
    </requirements>
    <files>
      <file label="exe" type="url">http://jobs.my.domain/job2</file>
      <file label="data" type="url">http://jobs.my.domain/more_data</file>
    </files>
    <invocation>
      <binary type="reference">exe</binary>
      <args>
        <arg type="literal">-f</arg>
        <arg type="file">data</arg>
      </args>
    </invocation>
  </job>

</jobset>
```

Looking at the last job, the invocation at the peer would be something like:

```
$ job2 -f more_data
```

Chapter 6

Command-line arguments

6.1 peer

All of the configuration options for a peer, except for `jail`, may be specified on the command line, rather than through a configuration file. If both are present, the command-line configuration will override the values in the file. The following command-line options are supported (with configuration file equivalents in brackets:

-c <client port> The port on which the peer will listen for client connections. [`client_port`]

-p <peer port> The port on which the peer will listen for connections from other peers. [`peer_port`]

-h <hostname> The name (or IP address) of the host on which the peer runs. [`host`]

-k <privkey file> The file containing the peer's private key. [`private_key`]

-x <certificate file> The file containing the peer's public key certificate. [`certificate`]

-X <CA certificate file> The file containing the grid owner's public key certificate. [`owner_certificate`]

-C <CPU speed> The advertised CPU speed of the host. [`CPU`]

-M <memory> The advertised RAM available on the host. [`memory`]

-S <disk space> The advertised disk space provided by the host. [`storage`]

-v Increase the verbosity level by one. This may be specified multiple times, and adds to the configuration file's value, rather than overriding it. [`verbosity`]

Note that all command line arguments may be specified in any order. If an argument is supplied multiple times, only the last one will be used.