NEOS Server 4.0 Administrative Guide[*]

Elizabeth D. Dolan[†]

Latest Update: May 30, 2002

Technical Memorandum ANL/MCS-TM-250

Abstract

The NEOS Server 4.0 provides a general Internet-based client/server as a link between users and software applications. The administrative guide covers the fundamental principles behind the operation of the NEOS Server, installation and trouble-shooting of the Server software, and implementation details of potential interest to a NEOS Server administrator. The guide also discusses making new software applications available through the Server, including areas of concern to remote solver administrators such as maintaining security, providing usage instructions, and enforcing reasonable restrictions on jobs. The administrative guide is intended both as an introduction to the NEOS Server and as a reference for use when running the Server.

[*]This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38, and by the National Science Foundation (Challenges in Computational Science) grant CDA-9726385 and (Information Technology Research) grant CCR-0082807.

[†]Industrial Engineering and Management Sciences Department, Northwestern University, and Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439; e-mail: dolan@mcs.anl.gov

Contents

1 Introduction **3**
1.1 Key Concepts 3
1.2 The Token Configuration File 4
1.3 The NEOS Comms Tool 5

2 Installing the Server **6**
2.1 Configuring and Building the Server 6
2.2 Reconfiguring the Server 7
2.3 Modifying Server Content 7

3 Running the Server **8**
3.1 Starting and Stopping the Server 8
3.2 Running the Server via Cron 8
3.3 Server Checks 9
3.4 Monitoring Server Traffic 10

4 Solvers on the NEOS Server **12**
4.1 Administrative Solvers 12
4.2 Registering a Solver 13
4.3 Taking Advantage of the NEOS Web Site 16
4.4 Hooking Solvers 17
4.5 Starting Communications with the NEOS Server 18

5 Server Implementation **20**
5.1 The Directory Structure 20
5.2 Submission Flow 21
5.2.1 The Receiver 22
5.2.2 The Initializer 23
5.2.3 The Parser 24
5.2.4 The Scheduler 24
5.2.5 The Job Scripts 25
5.3 The Socket Server 26
5.3.1 Interface with the Job Pipeline 26
5.3.2 Socket Communications Requests 27
5.3.3 NEOS Submission Tool Requests 28
5.3.4 Comms Tool Requests 30
5.3.5 Server Requests to the Comms Tool 31

6 Administrative Solver Implementation **33**

7 Web Site/Interface Implementation **35**
7.1 The Web Server Role in Submission Flow 35
7.2 Web Site Creation 36

8 The NEOS Server Administrator FAQ **38**

Acknowledgments **38**

1 Introduction

Consider hundreds of software applications with differing kinds of input data and potentially thousands of users, each needing some of these applications. The programming effort required for all software developers to write application service providers that would make their codes remotely available over the Internet would be prohibitive. Consequently, the only way for users to access many applications is to download and install the codes locally. In some cases this is the best solution, but in other cases the code either is not portable or is highly optimized for a given architecture. In these cases users might obtain an account on the host machine to use the software. Obviously, this solution is not scalable.

The solution that the NEOS Server provides is that of a general Internet-based client/server, providing a link between users and software applications. Any application that can be tweaked to read its expected input from one or more files and to write meaningful results to a single file can be automatically integrated into the NEOS Server. NEOS solvers are then available to users via all of the NEOS Server's interfaces. The user interfaces provided by NEOS are tailored to each particular solver based on information provided by the solver administrators, yet the interfaces still possess the same look and feel regardless of which solver the user is accessing. The advantages of this system are obvious in that users can freely browse the collection of solvers without having to learn how to use a new interface for each software application. Software developers benefit by easily adding their applications to a system that manages the complexities of client/server interactions for them.

In short, the NEOS Server handles interactions between a set of users and a set of solvers. A user submits formatted data to the NEOS Server through one of its interfaces. A *solver* reads its input from one or more files, processes the input, and outputs the results to a single file. For example, one solver may be a suite of Fortran code that minimizes an objective function. In this case the input could be a set of files containing the objective function and starting point source code, while the output could be a file containing the solution vector and other information associated with the solution process. The job of the NEOS Server is to connect the solver with the user's data and the user with the solver's results.

The process just described is typically referred to as client/server. Implementing client/server technology is straightforward, if somewhat tedious. Implementing a system abstract enough to handle the communications needs of a wide range of applications and users while enabling as many conveniences as possible presents some challenges. This guide introduces the NEOS Server as a way to meet these challenges. The guide begins with descriptions of the building blocks of the Server functionality and the steps necessary to install and run the Server and solvers. Descriptions become increasingly detailed as the guide progresses to the implementation of the NEOS Server in terms of submission flow and hints for administrative trouble-shooting and adaptation.

1.1 Key Concepts

The NEOS Server relies on the implementation of two main abstractions—that of data and that of services. The NEOS Server user interfaces currently consist of email, World Wide Web, socket–based NEOS Submission Tools (NSTs), and Kestrel (CORBA) interfaces. To avoid needless complexity, each interface must provide the NEOS Server with the user's data in a consistent manner. Indeed, the submission parser should have no need to distinguish among possible original user interfaces. Further, the parser should be able to accept various kinds of data to support a variety of application needs. To these ends, the NEOS Server has adopted a standard representation for all user data, described in Sections 1.2 and 4.2.

The Server's abstract implementation of services allows it to schedule all user jobs as service requests, handled through a *job_start* script. At a central level, the Server need not be aware of the type of service it is enabling or whether user jobs are served on the Server machine itself or on a remote machine. The NEOS Server assigns each job to a *job_start* script that executes application software via requests to a communications daemon assigned to the application and solver station. The Server can delegate the tasks of interpreting data and executing software to the (potentially remote) solver machines through the NEOS Comms Tool, introduced in Section 1.3 and detailed in Sections 5.2.5 and 5.3.2. Through abstraction and delegation, the NEOS Server can enable a wide variety of application services.

1.2 The Token Configuration File

Token configuration files are provided by solver administrators when they add their solver to the NEOS Server. Each line in a token configuration file specifies the name of an input file that the solver is expecting, along with tokens that can be used to delimit the data written to this file. A simple example configuration file helps to illustrate:

```
1st file::begin.a:end.a:A
2nd file::begin.b:end.b:B
3rd file::begin.c:end.c:C
```

Notice that there is one line for each input file the solver is expecting, namely, files A, B, and C. In addition, there is a set of tokens used to delimit the data coming from the user, and a label for the data (e.g., "1st file"). Users of graphical interfaces (e.g., NSTs, WWW) will simply see the label and a space for entering the location of their local input file. Email users are expected to place all of their data into one file and delimit input sections with the appropriate tokens. Cut-and-paste outlines for email submission are available via email and through a Web page. For example, a token-delimited submission might look like the following:

```
begin.a
  contents of 1st file
end.a
begin.b
  contents of 2nd file
end.b
begin.c
  contents of 3rd file
end.c
```

The reason graphical interface users are relieved of token delimiting their input themselves is that the interface does this for them before sending the submission to the Server. Kestrel users have a client that extracts their modeling environment's native format for NEOS submission, token-delimits the various types of data, and sends it to the NEOS Server. Kestrel users have no need to see the content between tokens, much less the tokens themselves. In the end, though, the NEOS Server always receives data in token-delimited submission format, regardless of the interface that sent it.

When the Server receives the submission file containing the contents of the user's three files, it must take this data and place each datum in the files expected by the solver (i.e., A, B, C). The Server uses the solver's token configuration file to decide what to do with this input. In this case, all data between the tokens "begin.a" and "end.a" gets saved to file A, and so on. The NEOS Server software packages the files, sends them to the machine housing the solver, and there unpacks the files. It then calls the solver, which in turn reads in the files A, B, and C, processes the data, and generates a result. The result is then returned to the user via the same interface used originally to submit the data.

1.3 The NEOS Comms Tool

The heart of client/server technology lies in specifying the types of communications available to the clients and servers. While less obvious than the relationship between users (clients) and NEOS (server), the dealings between the NEOS Server and the various solvers also represent a client/server model. In the early days of NEOS, the Server and solvers were all on the same system, so communication took place through files. Now, however, solvers are remote relative to the Server, for both security and modularity. Hence, solvers must also have an Internet interface to the Server, just as users do. This interface is referred to as the NEOS Comms Tool, which downloads the user files from the NEOS Server, invokes the solver application, and returns the solver's results to NEOS.

The NEOS Comms Tool is a small client/server application whose client is the NEOS Server and whose server is a daemon running on each solver station. The NEOS Server, upon receipt of a job for a particular solver, connects to the machine running the Comms Tool daemon for that solver, uploads the user's data, and requests invocation of the remote solver. The Comms Tool daemon, upon receipt of this message, downloads the user's data and invokes the application software. Intermediate job results can be streamed back over the connection to the Server and from there, in turn, streamed back to the user. The Comms Tool daemon sends final results to the NEOS Server upon completion of the job, and the Server formats and forwards the results to the user.

NEOS users and solvers are thus completely abstracted from one another via the NEOS Server. The advantages of this abstraction are immediate in that small changes in the solver applications leave the users unaffected. On the other hand, improvements to the NEOS Server in the center of this communication can benefit all solvers and users at once. An example of this advantage might be the addition of NEOS Server routines to handle data submitted to the Server with DOS or Macintosh file formatting. Through a single change to the Server, users gain the ability to submit jobs from diverse systems to any of the solvers. Solver users only need to know how to connect to the Server to submit jobs, and solver administrators only need to know how to make the Server aware of their solvers.

2 Installing the Server

The NEOS Server is intended to run on standard Unix-based platforms. Because almost all of the Server is written in Perl, the system must have Perl (version 5 or greater) installed. The NEOS Server relies on the `tar` and `gzip` facilities to package data for sending across the Internet. To use the email interface to NEOS, the system needs the `MH` mail facility and a `C` compiler installed.

Before proceeding to the configuration and modification of the Server, it helps to have answers to the following questions:

- What userid should run the Server? We suggest creating a new Unix account that will be dedicated to running the Server (e.g., the user `neos`). The Server can be run under a user's existing userid, but use of the email facility would lead to the Server treating all of the user's incoming mail as job submissions. We do not recommend running the Server as root either.
- Where should the Server be installed? We recommend unpacking the Server somewhere on the local hard drive of the machine hosting it so that the Server is not as prone to the difficulties of a network file system. A parent directory for the variable length files, including job submissions, logs, databases, and other records may be placed in a separate location. We refer to this directory as *server_var*.
- What interfaces should the Server support? We suggest starting with the Web interface. Then progress while installing the NEOS Server is clearly visible.
- If the Server supports a Web interface, what are the absolute directories and base URLs for the HTML and CGI pages? Web masters or system administrators should be able to provide answers (e.g., absolute directories /home/www/neos and /home/www/cgi-bin/neos with URLs `http://www.mcs.anl.gov/neos` and `http://www.mcs.anl.gov/cgi-bin/neos`).

After deciding on the userid and home directory for the Server, installation begins with becoming that user, going to that directory, copying the Server package there, and executing the following commands:

- `gzip -d server-4.0.tar.gz`
- `tar -xfv server-4.0.tar`

This process should create the directory server-4.0. For the remainder of this manual, Server file locations given are assumed to reside under server-4.0, unless they are listed as being in *server_var* or some other directory supplied by the Server administrator in the interactive Server configuration process. The next section will discuss configuration of the Server.

2.1 Configuring and Building the Server

Configure and build the Server using the userid that will be running the Server. Then `cd` into config/, and execute `make`. The Makefile will launch the configuration script server/bin/*config.pl* that will step through the entire configuration process interactively. After all configuration parameters have been set, the script builds a set of front-end administrative scripts that are used to start, stop, monitor, and maintain the Server. A successful build looks something like this

```
Building Server:

building checker script...ok
building monitor script...ok
building restart script...ok
building kill-server script...ok
building cleaner script...ok
building report script...ok
building dailyreport script...ok
building weeklyreport script...ok
compiling some...
make[1]: Entering directory '/neos/server-4.0/server/src'
cc address.c -o ../bin/address
make[1]: Leaving directory '/neos/server-4.0/server/src'
generating crontab for backup and reports...ok
```

All of the configuration parameters are saved to the file server/lib/config-data.pl. The *config.pl* script rewrites the contents of that file in the format of a shell script and appends a call to the corresponding Perl script under server/bin/ for each administrative script built. The administrative scripts that can be executed on the command line and used to run the Server are placed in bin/. If these scripts fail to execute correctly, the Server may need to be reconfigured.

2.2 Reconfiguring the Server

If the Server is already running, be sure to stop the Server before reconfiguring by executing the administrative script bin/*kill-server*. Then reconfigure the Server simply by running `make` from within the directory config/. The old configuration has been saved and is now represented as the default. To remove the old configuration, run `make clean` before running `make`. Running `make` and `make clean` does not destroy any of the Server state information, archived jobs, or logs saved in the directory *server_var*. Executing `make allclean` removes these files before reconfiguration, which is equivalent to installing a fresh Server. After reconfiguring the Server, restart the Server with *bin/ restart*.

2.3 Modifying Server Content

Most text and HTML content used by the Server can be modified to suit the particular installation of the Server. Solver types may be added or removed in the file *server_var*/lib/solver_tree, which contains a list of solver categories (of which there must be at least 1). File *server_var*/lib/feature_tree provides an alternative primary grouping scheme for the solvers in the same file format. Features from this file are added as checkbox options to the ADMIN:ADDSOLVER for solver registration, and the grouping shows up as an alternative solvers Web page server-solvers-in.html.

Changes to files under server/lib/html/ and server/lib/txt/ modify the content of the Web site pages or the text information files associated with the Server. The Server must be rebuilt to reflect changes to these files in the automatically generated HTML pages and CGI scripts. Rebuilding the Server can be accomplished by executing `make` in the directory config and accepting all of the defaults.

3 Running the Server

If the Server has been configured properly, starting the Server is as simple as typing a single command. Keeping the Server running and checking that it is executing as intended requires a bit more effort.

3.1 Starting and Stopping the Server

To manually start the NEOS Server, execute the script bin/*restart* on the machine where the Server has been configured. Restarting should produce output similar to the following:

```
killing receiver...
killing scheduler...
killing socket-server...
starting NEOS Server receiver...
starting NEOS Server scheduler...
starting NEOS Server socket-server...
```

Assuming everything started correctly, the command `ps -o "pid args"` should generate output similar to the following:

```
5 /bin/perl /neos/server-4.0/server/bin/receiver.pl
7 /bin/perl /neos/server-4.0/server/bin/scheduler.pl
8 /bin/perl /neos/server-4.0/server/bin/socket-server.pl 3333
```

These are the three primary daemons that compose the Server. The *receiver* script waits for incoming email, Web, NST, and Kestrel jobs, passing these along to the *scheduler*. The *scheduler* schedules these jobs on solver workstations by contacting the remote Comms Tool daemon running on the workstation. The *socket-server* serves all TCP/IP socket traffic between the NST and the Server and the Comms Tool and the Server. If any one of these three scripts is not running, the NEOS Server will not function correctly.

To manually stop the Server, execute the script bin/*kill-server*. The following output should appear:

```
killing receiver...
killing scheduler...
killing socket-server...
```

3.2 Running the Server via Cron

Configuring/building the NEOS Server creates the file config/crontabfile. This file contains crontab entries for the scripts *checker*, *cleaner*, *dailyreport*, and *weeklyreport*. The *checker* will restart the Server in the case of machine reboots or Server crashes. If added to the user crontab, usage reports created by the *dailyreport* and *weeklyreport* scripts are mailed to the Server administrator and the email address for Server comments at the appropriate time intervals. The crontabfile also handles calling the *cleaner* to archive old job directories, delete out-of-date Web results pages, and make provisions for strangely aborted jobs. While the reports are optional, and restarting the Server is necessary only occasionally, the Server administrator should make a point of running the *cleaner* regularly.

To enable these scripts via `cron`, the cron entries in config/crontabfile need to be added to the userid's current crontab. If the crontab is empty, run `crontab crontabfile` to install the new crontab; otherwise, edit the crontab, and manually add the new entries. On most systems, `crontab -e` opens an editor for editing the crontab directly.

Via the *checker*, the cron daemon will restart the Server after system reboots or in the event that the Server is not responding. The cron daemon invokes the checker, which connects to the *socket-server* daemon and waits for a response. If the *socket-server* daemon is not running (or does not respond in time), then the *checker* will assume that the entire NEOS Server is dead, in which case it will kill the remaining Server daemons and exit. Upon the next invocation of the *checker*, the Server will be restarted via `bin/restart`.

The cron daemon does start processes as the user identity that invoked the `crontab`, but cron jobs entail minimal initial running environments. Section 4.4 contains more on the limitations of cron for solver administrators. The Server administrator should know that if the Server is restarted by the cron daemon, the `ps` command will no longer show the running daemons (as they no longer have a controlling terminal). Check the manual pages on the system to find the option to `ps` that shows processes without controlling terminals. A few other tests can be run to check the Server.

3.3 Server Checks

Once it is determined that the daemons are all running, several other tasks will help to show that everything is working correctly. A simple job submission from any interface can test most of the main Server scripts. Submissions via email and from the Web site interface, if enabled, provide assurance that the special features associated with these interfaces are operable. Section 3.4 begins the discussion of additional tools to help debug any problems.

In some ways the NST interfaces represent the best initial test of the Server because a successful NST job completion indicates that socket requests are flowing through the *socket-server* daemon and the *receiver*, *initializer*, *parser*, *scheduler*, and *job* scripts are at least minimally functional. An NST submission first requires the installation one of the NSTs (Tcl/TK or Java, currently). More information on installing the Tcl/TK client tools is given in Section 4.5. If the tool's graphical user interface appears, then the *socket-server* script is generally functional. A blank *Adding/Modifying a Solver* job submission from the NST tests the functionality of the job requests to the *socket-server*. The solver should return a list of the categories in *server_var*/lib/solver_tree.

A simple email interface job tests the Server by mailing the following message to the appropriate address:

```
help admin:addsolver
END-SERVER-INPUT
```

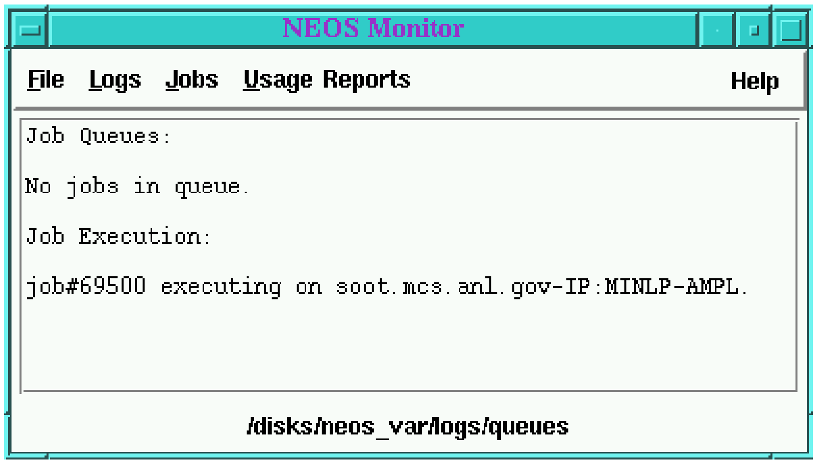
Requests for email interface help are scheduled like other jobs, so results from this submission should indicate the functionality of the email interface to the NEOS Server. Additionally, the help returned includes a template for mailing *Adding/Modifying a Solver* jobs.

If the Server's Web interface is enabled, check the URL to ensure that the Web site is visible. If nothing appears at the expected URL, the absolute directory should be checked. If the pages are there, the Web master may be able to determine the correct URL. If the directory contains no Web pages, the write permission to the parent directory may need to be adjusted before rebuilding the Server. Assuming the Web site has been built, all of the solver categories added to the *server_var*/lib/solver_tree should appear on the server-solvers.html Web page. Under the `WWW Form` link on *Adding/Modifying a Solver*, a blank job submission tests most features of the Web site CGI scripts. If the other interfaces are working while the Web interface is not, the Web master may be able to help with system-specific pointers about the use of CGI scripts.

Should any of the above tests fail, more specific hints as to the problem may be found by monitoring the Server logs and database. The socket log can be especially helpful when the Server fails to restart properly.

3.4 Monitoring Server Traffic

While all of the Server logs and databases are essentially just files, we do provide an extra `monitor` tool to view and query them. The monitor is not a requirement for running the Server, and the system must have `wish` and `Tcl/TK` installed for the monitor to function. To start the monitor, run the bin/*monitor* script. The script should open a graphical user interface (Figure 3.1) with menu options to view the various Server logs, make queries to the master database, retrieve job submissions and results, and create usage reports.



```
Job Queues:

No jobs in queue.

Job Execution:

job#69500 executing on soot.mcs.anl.gov-IP:MINLP-AMPL.
```

/disks/neos_var/logs/queues

Figure 3.1: NEOS Monitor

All Server activity is logged in the directory *server_var*/logs. There should be a file for the *receiver* daemon, one for the *scheduler*, one for the *socket-server*, and one for the *checker*. Additionally, there is the file queues, which is not really a log but lists the currently executing jobs and jobs in the scheduler's execution queues.

Assuming that everything was configured correctly and the Server started, the receiver log should contain output similar to

```
receiver: My Server restart on Aug 8 16:38 PDT 1999.
```

The scheduler log should contain output similar to

```
scheduler: Restart on Aug 8 16:38 PDT 1999.
```

The socket log should contain output similar to

```
My Server accepting connections on opt.mcs.anl.gov port 3333.
```

The checker log (if the Server was started via cron) should contain output similar to

```
client.pl: ERROR: connect: Connection refused
checker: killing My Server processes on Aug 8 16:38 PDT 1999.
checker: restarting My Server on Aug 8 16:38 PDT 1999.
```

and the queues log should look like this:

```
Job Queues:

No jobs in queue.

Job Execution:

No jobs executing.
```

A couple of hints about these logs are worth a Server administrator's attention. If the Server administrator ever needs to completely abandon jobs listed in the queues (because of some catastrophic failure), the Server must be killed and the queues file deleted along with every file under the *server_var*/proc/stations/ directory before the Server is restarted. The Server must be restarted if the logs are edited for any reason (such as size reduction). Most of the NEOS Server daemons are unable to write to altered logs, eliminating a good source of trouble-shooting data if the Server is not restarted.

One common problem that can cause Server tests to fail is easily spotted in the socket log. When the NEOS Server is not killed correctly, it may show the restart message

```
Cannot bind to port 3333!
```

After multiple attempts to bind the port, the NEOS Server will send the Server administrator an email message warning that the Server could not be started. If changes to the Server configuration do not appear to be functioning after a Server restart, the socket log should indicate whether an older version of the *socket-server* might still be running. In fact, it is not a bad idea to check the socket log every time the Server is restarted.

4 Solvers on the NEOS Server

One of the conceptual challenges presented by the the creation of a generic application service provider involves the interface the NEOS Server supplies for communicating with a wide variety of software applications. The interface should be flexible enough to handle many different input, output, and systems demands on the part of the applications while adding enough support to make the process of adding an application to the available pool worthwhile. Our system is designed to allow solver administrators at distributed locations to take responsibility for solver applications that run on their sites. The solver administrators, like other users, need not have accounts on the same system as the NEOS Server. They register their solver with the NEOS Server using an administrative solver, write an application wrapper that can be executed on their site, and start a communications daemon on each solver station they make available. The NEOS Server is designed with few requirements to make an application accessible through the system but with many options for customizing and enhancing solvers.

In this section, we offer solver administrators an explanation of the steps involved in adding applications to the NEOS Server. The NEOS Server includes a collection of built-in administrative solvers, which are used in registering new solvers. The information required for registration is detailed along with supplementary data helpful to solver administrators for informing users about the purposes and requirements of their software. Particular attention is paid to the role solver registration plays in building the NEOS Web site. The necessary execution requirements for software applications are presented in Section 4.4 on hooking solvers to NEOS. Finally, Comms Tool daemon startup on the solver stations completes the picture of building a NEOS Server by adding solvers. Along with these sections of the guide, the NEOS Server administrator can point solver administrators to the *Solver Administrator FAQ*, `admin_faq.html`, that is built with the NEOS Web site.

4.1 Administrative Solvers

The NEOS Server comes packaged with a suite of administrative solvers, which function in much the same way as other solvers. These solvers include *Adding/Modifying a Solver* (ADMIN:ADDSOLVER), *Enabling/Disabling a Solver* (ADMIN:STATUS), *Kill Job* (ADMIN:KILL_JOB), and the *NEOS Help Facility* (ADMIN:HELP). The first two solvers are used strictly by solver administrators, while the third may also be employed by a user with the appropriate job password. The help facility provides Email interface information to a user about a particular solver.

The most important two of these administrative solvers provide the mechanisms for adding and deleting solvers. The ADMIN:ADDSOLVER solver allows NEOS to automatically increase its collection of solvers. The basic purpose of the ADMIN:ADDSOLVER is to generate all of the user interfaces to solvers based on the information provided by the solver administrators. For example, when a solver is added, ADMIN:ADDSOLVER will enter the solver identifier in the NEOS Server's list of available resources and automatically create the solver's WWW interfaces (both HTML and back-end CGI). The ADMIN:STATUS solver allows the Server to shrink its solver collection. If solver administrators ever wish to temporarily disable their solver or completely delete their solver from the Server, they can just submit this job request to the ADMIN:STATUS solver, which deletes the solver identifier from the list of available resources and, using the WWW example, either temporarily disables the links to the solver's Web pages or completely deletes the Web area for the solver.

Solver administrators can read more about the other administrative solvers in Section 6. That section on the implementation of administrative solvers describes how ADMIN:HELP takes advantage of email-help information registered with the solver. *Kill Job* should be tested by solver administrators before they decide whether to enable it for their solvers, as discussed in Section 4.5. Neither of these solvers is crucial to the NEOS Server, but they do offer features helpful when the Email interface is enabled or when remote solver stations are able to completely kill off application processes on demand.

The ADMIN solvers available to solver administrators allow them the greatest possible flexibility in determining the interactions between the NEOS Server and their solver without requiring direct access to the machine hosting the NEOS Server. The NEOS Server makes its administrative solvers available through all of the interfaces provided for regular solvers so that solver administrators have no need for an account on the NEOS Server machine. These interfaces are built based on the same type of token configuration files required for other solvers. The main difference between the administrative solvers and other solvers is simply that the administrative solvers come packaged with the NEOS Server.

4.2 Registering a Solver

Solver administrators can register their solvers with the NEOS Server through any interface by using the administrative solver *Adding/Modifying a Solver* (ADMIN:ADDSOLVER). When a solver is registered, it is automatically enabled, meaning the NEOS Server may attempt to relay incoming submissions to the solver. Through the *Enabling/Disabling a Solver* (ADMIN:STATUS) solver, administrators may temporarily disallow submissions to their solvers and enable the solver again later. The registration form submitted to ADMIN:ADDSOLVER contains entries for the following information.

- **Solver Type**

  New solvers must be placed under an existing NEOS Server solver category. Submitting a job to ADMIN:ADDSOLVER with a blank or incorrect solver type will trigger the Server to return a list of available solver categories. The ADMIN:ADDSOLVER expects to receive the abbreviated type name (e.g., `misc`, not `Miscellaneous`).

- **Solver Group**

  The NEOS Server currently creates two Web page listings of the solvers available. The main server-solvers.html page uses the solver type as the primary category for organization with the solver group as the secondary category. A solver group is listed under the proper solver type category with links to the separate Web pages for each interface given on one line. The link labels that differentiate separate interfaces within a group are determined by the features registered for that interface. While many entries may fall under a solver type, it is expected that members of a solver group will be few (i.e., no more than will fit on a line of text of a reasonable length). To be listed, an interface must have one or more features unique to it among all members of the group. Kestrel solvers, which are not listed on the Web page, can use a solver group and feature to make sure that the Web page of any normal solver that matches both has a link to information on the Kestrel client.

- **Solver Identifier**

  The NEOS Server identifies solvers through their type, identifier, and password. A solver identifier should be chosen as a single word without any special characters, guaranteeing only that letters, numbers, and underscores will function correctly. A solver identifier must be unique within its solver type category.

- **Solver Name**

  Each solver is also given a full name, which may contain spaces and other special characters. The NEOS Server generally displays the solver name in its various interfaces but uses the solver identifier internally.

- **Solver Password**

  The password entry represents an attempt to ensure that only authorized users or the Server administrator is able to reconfigure solvers. This password is not extremely secure. Solver administrators should **not** use their regular account passwords.

- **Contact Person**

  The ADMIN:ADDSOLVER scripts insist that solver administrators give an email address so that they can receive error messages when their solver misbehaves or is abused. The address is not currently checked for validity but may be in future releases of the NEOS Server package.

- **Workstations [jobs allowed]**

  The solver administrators must also give a list of workstations where their solver will run. The NEOS Server must have the full machine address of each solver station. Optionally, the administrator may enter a positive integer on the same line following the machine address that indicates the number of jobs submissions that can be run on the solver station concurrently. An entry might look like the following:

  ```
  harkonnen.mcs.anl.gov 3
  ```

- **Token Configuration**

  The solver administrator gives the full path to a file containing the tokens that will delimit job submission data for the solver. Each line in the configuration file contains five colon-separated entry fields. The first field provides the data label that appears on the Web and submission tool forms for the solver. The second field specifies the type of data or the default value of the data. The data types that may be specified are `TEXT`, `BINARY-ON`, `BINARY-OFF`, and `RADIO`. The `TEXT` type referred to here provides a large text entry field with scroll bars in the graphical interfaces. Any other entry will be treated as a default value for a text variable or file name and appears only in the NST GUI. The third and fourth fields specify beginning and end tokens. If the end token is `NULL`, the data type will be a simple text variable (without scroll bars in the GUIs), and the submission format will require that the value be given on the same line as the beginning token. If the beginning token for a file name data type (i.e., an unspecified type with an end token) ends with `BINARY`, the file uploaded to the NEOS Server will be preserved with no mangling. The fifth field gives the name of the file where the data will be written on the solver station. Lines in the token configuration file appear as follows.

  ```
  Objective source: fcn.c: begin.func: end.func: FCN
  Language: RADIO .C,c .Fortran,fortran: lang: null: LANG
  Number of variables: : NumVars: NULL: NumVars
  Use alternative algorithm?: BINARY-OFF: AlgB: null: AlgB
  Job label: TEXT: BEGIN.COMMENT: END.COMMENT: COMMENTS
  ```

  The token configuration file is the only file necessary to register a solver. It allows various types of data to be written to specified files. In this example, the Server parses incoming submissions to the solver and writes the source code for the function evaluation to FCN. It places the text value `c` or `fortran` in the LANG file and another integer text value in NumVars. The `BINARY` data type accepts values of `yes` or `no` and writes them to AlgB. Finally, a text field for comments saves its data in the file COMMENTS. The begin and end tokens are not case sensitive, but files appear by exactly the fifth field names in the directory where the solver driver or executable is invoked. A fully token-delimited submission to a solver with identifier `mine` and type `misc` registered with this token configuration, whether formatted by the interface or directly by the user as an email submission, might appear as follows.

  ```
  TYPE MISC
  SOLVER MINE

  begin.func
  double fcn(double *x, int n) {
      double f = 0.0;
      for(int i=0; i<n; i++)
  ```

```
        f += x[i] * x[i];
    return f;
}
end.func

lang = c
NumVars  = 10
AlgB     = no

begin.comment
A simple example.
end.comment

END-SERVER-INPUT
```

The NEOS Server makes no effort to check the contents of the input files for validity. This task is delegated to the individual solver drivers (Section 4.4). The files and other data described next are optional, and further descriptions of the Web-related entries follow in Section 4.3.

- **Usage Restrictions**

  Solver administrators may limit the number of jobs sent to their solver stations. The usage restriction file contains lines of the form

  `<users> <minute|hour|day|month> <limit>`.

  The following is an example:

  ```
  #max                hour  20
  #max_any_one_user   day   30
  #max_any_one_domain month 100
  ```

- **Email Help**

  The Email help file is sent to users who request help via email from the ADMIN:HELP facility. It should contain a template with the data tokens for Email interface submissions.

- **Submission Tool Help**

  The submission tool help file is returned to users who submit a request for help from a solver's submission form as accessed through an NST.

- **WWW Help**

  The Web help file contains the text that appears on the solver's Web submission page. The text may contain HTML tags for better formatting. In addition to the usual HTML tags, this file should contain one `<NEXT_ENTRY>` tag for each entry in the token configuration file. A description of the data necessary for each entry may precede the `<NEXT_ENTRY>` token. If the solver administrator supplies no WWW help file, the Web submission form is still created, with only the labels from the first field of each token configuration line to guide the user.

- **WWW Samples**

  The Web samples file gives descriptions and URLs for sample solver submissions. The submissions must be formatted as would an Email submission. The WWW samples file contains two lines for every sample. The first gives a description that may contain HTML tags. The second is the full URL to a Web-accessible sample. The sample page under the solver's home page on the Web site will enable users to make practice submissions. To provide easy viewing of the sample, a hyperlink may be added to the description.

  As an alternative to the two-line format, which results in a list of samples, the samples may be formatted in a table. For example,

```
<COLUMNS 3>
Problem
Algorithm A
Algorithm B
<a href=''ftp://ftp.some.site/exmpA.txt''>Production</a>
http://www.some.where/exmpA.txt
http://www.some.where/exmpB.txt
```

would create a table with examples for algorithms A and B.

- **WWW Abstract**
  The WWW abstract file is displayed as the solver's home page within the NEOS Web site. The text may contain HTML tags as would appear below the `<BODY>` token. Knowledge of HTML is not necessary, but plain text may not be formatted as nicely.
- **WWW About URL**
  The Web background URL text field should contain the full URL to a Web page providing more information about the solver if such a page exists.
- **Features**
  The alternative Web listing of solver interfaces, server-solvers-in.html, sorts entries primarily by their features registered. It is expected that many solvers may be registered with the same feature. The Server administrator can offer a standard selection of features by entering them in the *server_var*/lib/feature_tree. Solver administrators can also write their own special features in the space provided for other features. Features are also used in the main solver listing Web page to distinguish between members of a solver group. Therefore, separate solver group members must be given unique features.

4.3 Taking Advantage of the NEOS Web Site

When registering a solver, the NEOS Server offers the opportunity to craft a mini-Web site devoted to that solver. The NEOS Server Web site construction allows for a separate group of pages for each solver registered on the system. The mechanisms that allow automatic solver Web site construction and enable Web interface are discussed in more detail in Section 7.

Each solver has a descriptive homepage, a sample submission page, a user comments page, a page of instructions for email use with a solver-specific submission template for emailed job requests, and a form for submitting jobs from a Web browser. The NEOS site links these pages via a hyperlink to the solver homepage from a central NEOS Web page listing all solvers by category.

When solver developers register their solver on the NEOS Server, they must supply at least a solver type, identifier, name, contact email address, and a token configuration file. From this information alone, the NEOS Server can build the solver's Web submission form and the pages describing the email template. The solver's name will appear on the list of available solvers with other solvers of that type. In a technical sense, these tools are enough. Unless all of the Server's intended users are already familiar with the solvers offered, though, more information is critical.

Sample Submissions

The quickest introduction a user will find to submitting NEOS jobs is through the sample Web submissions. Solver administrators should pick a few appropriate trial submissions, prepare them in token-delimited format for the NEOS Server, and place them in a Web-accessible area. The extensions on the files should indicate plain text so that the Web browser will not try to render them as if they were HTML when users view them in a browser. The administrators compile a file outlining the sample submissions available with the simple format of one line for the example's label followed by a line containing only the URL of the sample file. The labels may include hyperlinks to the sample text. By providing the user with working code, the sample submission Web page makes the use of token-delimited entries clear. It also allows the skeptical to prove to themselves that the NEOS Server is up and running, without investing much time in learning the system.

Solver Homepage

The registration form asks for an optional WWW abstract. This plain text description of the solver and its usage on the Server is incorporated into an HTML body to create the solver's homepage on the NEOS site. The process of creating solver homepages around the developers' supplied abstracts allows a similar look and navigational feel throughout the solvers offered. The name of the solver becomes a link to off-site Web pages when solver administrators supply a background URL. A *Sample Submissions* link steers users to the examples for each solver, and button links for sending in `Comments and Questions` or for returning to the NEOS Server `Home` are woven into each solver homepage as well.

For a more sophisticated homepage, HTML tags may be included in the developer's abstract text as they would in the `<BODY>` section of a normal HTML file. Because Web browsers will interpret the abstract as part of an HTML page, certain special characters must be delimited as for HTML. For example, the `<` symbol is denoted `&lt;`, the `>` symbol is `&gt;`, and the ampersand symbol is `&amp;`. Often in practice the abstracts are merely pasted from the solver developer's descriptive Web page, requiring no changes. Again, the solver administrators may devote only as much effort as they desire to registering the solver. Even if they submit no abstract whatsoever, the NEOS Server still creates a minimal solver homepage with the appropriate links.

Web Interface

The same philosophy is true of the Web submission interface. If solver administrators choose not to submit a Web help form for the solver during registration, the Web form for job submission will be built based on the token configuration file. If administrators want to offer more detailed explanations or examples for the form fields, they need only create a text file with a `<NEXT_ENTRY>` token delimiting the desired location within the text of each entry in the token configuration file. The resources of HTML hyperlinks are available to administrators, but they have the option of placing their solver on the NEOS Server with little effort just to see whether their software will be useful to a larger community. Administrators can always reregister their solver, adding details over time.

4.4 Hooking Solvers

To run an existing application via the NEOS Server, the application must be able to extract its input from a predefined group of files and send its output to a file called job.results, also in the current working directory. Generally, some sort of driver script is required to handle these tasks. Any executable script that can start the application software so that it receives the user input and writes output to job.results in the current directory is acceptable. A few points warrant consideration when writing such a script.

When the NEOS Server sends a job to a Comms Tool daemon, the daemon creates a temporary working directory for the job. The user-supplied files are unzipped in that directory and will be named as specified by the fifth field of each token configuration entry. These user files will be local to the driver script when it is invoked, but the driver script should not rely on relative locations for any other files. The driver script should state absolute paths to the application software and supporting files. Also, full path information for environment variables like the `PATH` should be set if the Comms Tool daemon is started with the `Cron` option, which has an extremely limited environment and does not check for any shell run command files (e.g., .cshrc). If the administrators' solver driver works when they start it but begins to fail later, these issues should be checked first in debugging.

Because applications may fail, it is in the solver administrators' interest to ensure that the user receives as much information as possible during job execution. Because users have intermediate results streamed into their Web browser or NST in approximate real time, the intermediate output from the application should be written to the Unix standard output without any buffering. The driver may need to flush output buffers explicitly to accomplish real-time updates and avoid the impression that the application is stalled. The final results should also be flushed as they are written to job.results so that failed applications provide the best possible hints about problems encountered. A NEOS solver that offers a user comment field should write the comments to job.results before the driver invokes the application. Users who write comments to label their submissions can then identify a job in case the application software fails, is killed, or times out. Execution information written to job.results can be overwritten with summary data by the driver after the job has completed successfully. Simply renaming the application's default output file at the end of execution, though, may mean that no job.results file is created for terminated runs, failing to meet the Server's one requirement of solver drivers. The user is more likely to resubmit jobs verbatim for which they receive no results than jobs that return error messages, and administrators will receive warning messages from the NEOS Server when their solver fails to create job.results.

While output to job.results is technically all that is required of a solver driver by the NEOS Server, the sophistication of the driver script greatly determines the overall functionality of the application. The manner in which an application reads options affects the amount of control users have over the application's execution. One common task of the solver driver is to specify command line options for the application, depending on the user input. An application that reads in an options file may require less effort on the part of the driver than an application that accepts only command line options. Depending on the options available, though, it may be a good idea for the driver to scan the user options file before the application is executed regardless of whether the application can read the options file itself. In this way, solver administrators can check that no unreasonable options are being leveraged by the user.

The solver driver script also serves to limit functionality not explicitly handled by the NEOS Server or via an option in the solver registration or Comms Tool startup. Not only should reasonable options be enforced, but user code that will be executed on a solver station should be scanned for potentially harmful commands. Running the Comms Tool via the account of a specially limited user can eliminate many of the potential risks. Alternatively, administrators may be able to exploit special options maintained by some applications defining restricted use by, for example, disallowing shell calls or file I/O. When solvers are distributed across different systems with varying policies, often the only way to gain any access to a site is by imposing specific limitations on that access. The next section discusses the technical mechanism by which the NEOS Server accesses and coordinates with other user identities on remote solver stations, including some options to limit the application execution.

4.5 Starting Communications with the NEOS Server

The NEOS Server supports a Perl implementation of a Comms Tool using TCP/IP for Unix. More information about the specific requests that can be made between the Server and Comms Tool daemon can be found in Section 5.3.2. While the Server specifically supports some implementations of a Comms Tool, others could be written to communicate with the NEOS Server from other platforms if they handled all of the requests mentioned in Section 5.3.5 and correctly made use of the requests available to them as described in Section 5.3.4. The information given next pertains to the Comms Tool implementations supported by the Server and their Tcl/TK graphical user interface for Unix systems.

To take advantage of the NEOS Server-supported Comms Tool, solver administrators must first install the Unix client on their site. They should download the current *client-version* package for their system and unpack it in the location of their choice. The client should create a client-

version/ directory with a Makefile in it. Solver administrators should check the Makefile to ensure that the machine name and port for the NEOS Server are correct and then `make` the client. This installation process should create *submit* and *comms* scripts under the directory client-*version*/bin/. The *submit* script launches the Tcl/TK NST, and the *comms* script launches the Tcl/TK Comms Tool interface. If the Comms Tool graphical interface does not load, then either the version of the client in incompatible with the solver administrators' system and another should be tried, or the client could not establish a socket connection with the NEOS Server. Error messages should help to determine which scenario is the case.

The solver administrator must start a Comms Tool daemon on each of the solver stations listed in the registration process. Communications cannot begin unless the solver is properly registered on the Server. Because the NEOS Server communicates directly with the Comms Tool daemons via a socket connection, no passwords are required. However, systems that rely on substantial security, such as those behind a firewall, may not allow normal users to bind to a socket for communication. Also, while the Tcl/TK Comms Tool interface expects that administrators can start Comms Tool daemons on all of their solver stations from one machine, in practice administrators may need to log on to each solver station separately, giving a password, to begin communications. The Tcl/TK Comms Tool can interface across machines using `rsh` or `ssh`; more information on the use of `ssh` and other aspects of the client tool is available in the *Solver Administrator FAQ*.

When solver administrators first execute the *comms* script of an installed *client-version* tool, the script creates a directory named .comms under the administrators' home directory on the system. Alternatively, the Makefile for the *client-version* tool may be altered to set an environment variable, `CACHE_HOME`, which will be the parent directory of .comms. Logs of communications between the Comms Tool daemons and the NEOS Server *socket-server* are stored under the .comms directory, and the temporary working directories for each job are created under this directory. The process identifiers for running daemons and the port associated with each are listed, and backups of the commands that start the Comms Tool daemons are also stored here.

Information required by the Comms Tool to start a daemon includes the full name of the machine hosting the NEOS Server and the port on which the *socket-server* is listening. Solver administrators must be able to identify their solver by type, identifier, and password and have access to the workstations that will serve as solver stations. An email address is required, as is the absolute path to the solver driver script discussed in Section 4.4.

The Comms Tool offers numerous options to customize the execution behavior of solver applications. Solver administrators may choose to change the default time limit, which will work effectively only if their operating system supports full use of process groups and the application software does not make any changes to the process group id of a job. The same restrictions apply to whether a solver's jobs may be labeled `Killable` when starting communications. The file size limit field may be adjusted to limit the size of any file created by the application, including all intermediate and results files, on systems that support the `limit` command. When selected, the `Notify` option requests that mail be sent to the contact address provided each time a job submission arrives. The `Debug` option ensures that any messages sent to the standard error stream of the solver are piped back to the user. `Save` prevents the Comms Tool daemon from deleting working directories when a job is completed. The `Cron` selection attempts to start a cron job to check occasionally that the Comms Tool daemon is running on the solver station and restart it if necessary. If solver administrators wish to disable this feature after communications are started, the `Disable Crontab` button does so. Solver administrators should normally disable the crontab for their solver before killing the Comms Tool daemon. Otherwise, the cron daemon will continually attempt to restart communications with the NEOS Server. More information on a particular version of the Comms Tool can be accessed through the *Help* button on the Comms Tool GUI.

5 Server Implementation

The remainder of this document is not intended for the casual reader interested only in the goals and use of the NEOS Server. Rather, we describe the software behind the Server for NEOS administrators who encounter problems, have special needs, or would like to prepare themselves for any eventuality.

5.1 The Directory Structure

The NEOS Server 4.0 package, when unpacked, places all of its contents in a directory called server-4.0/. The directory server-4.0/ contains three subdirectories: bin/, server/, and config/. When the Server is installed, it must also have access to some directories specified by the user, including *server_var*, a directory for Web pages, and a directory for Web CGI scripts, which may reside on separately mounted drives. Most of our discussion of NEOS Server implementation focuses on files found in the subdirectories of server-4.0/.

The directory bin/ contains front-end administrative shell scripts that make calls to some collection of Perl scripts within server/bin/ after the shell environment has been set appropriately. A glance at the contents of any of the scripts in bin/ reveals that they are all close to identical with the exception of the last line that executes one of the Perl scripts in server/bin/. These administrative scripts are all built during the configuration of the Server; hence, bin/ is empty when the Server is first unpacked.

The directory config/ contains the Makefile for installing the Server. The crontabfile containing entries for use by the cron daemon is built within config/ as part of the installation process.

Directory server/ is the only one at its level that is also a parent directory. Its subdirectory server/src/ contains one C source file, address.c, along with a Makefile that is executed during Server configuration if the Email interface is enabled. The directory server/bin/ contains Perl scripts and other executables called directly by the Server after its environment variable values have been established during configuration. The most extensive subdirectory of server/, server/lib/, acts as parent to most of the files that define the content, rather than behavior, of the Server.

The server/lib/ directory contains files generally considered read-only by the Server. This directory does contain templates for executable files, but these are not executables directly related to the Server, as is the case with all executables in server/bin/. For example, the scripts under server/lib/Perl/ are downloaded to the remote solver stations and called by the Tcl/TK Comms Tool daemon provided with the Server. The scripts under server/lib/TclTK/ are downloaded by the Tcl/TK clients, which are packaged separately but supported by the Server, and executed on users' local machines to build the graphical interfaces for the NST or Comms Tool. Similarly, the Server keeps an archive of Java classes for the Java NST under server/lib/java/ so that the *socket-server* can send them to the local machines of Java NST users in need of updated client versions. The exceptions to the read-only rule lie under server/lib/admin_solvers, where files ending in the suffix .server are rewritten with Server-specific data to a file in the same directory, named without the suffix. If the Server administrator wants to change the content of some aspect of the Server for which no option has been offered during configuration, server/lib/ represents a good place to look for the appropriate files.

The user-supplied directory *server_var* contains files and directories created, expanded, and deleted in the course of Server operation. This space includes directories for databases, logs, jobs, and process ids of running processes (proc), a spool directory for incoming jobs (e.g., from the Web server), a lib directory containing mostly individual solver registration information, and a tmp directory for use by the various Server processes.

The tables of files throughout Section 5 list and briefly describe most of the scripts associated with the NEOS Server. The executables mentioned in Tables 5.1, 5.2, and 7.4 reside under server/bin/. The scripts in Table 5.1 represent the guts of the administrative programs found in the bin directory and discussed earlier in Section 3. The Server scripts and executables in Table 5.2 are dealt with in

Table 5.1: **Administrative Scripts**

Script	Purpose
checker.pl	invoked by bin/checker to ping server
cleaner.pl	invoked by bin/cleaner to remove/archive jobs
config.pl	called by config/Makefile to configure Server
dailyreport.pl	invoked by bin/report, calls report.pl
monitor.tk	invoked by bin/monitor to view logs/database
query.pl	queries master database
report.pl	invoked by bin/report to generate usage reports
restart.pl	invoked by bin/restart to kill and restart server
weeklyreport.pl	invoked by bin/weeklyreport, calls report.pl

much detail in Section 5.2. The scripts for Web site construction are further described in Section 7.2. The CGI script templates in Table 7.3 can be found under server/lib/cgi/. The location of these CGI scripts after the Server-specific information has been added to the templates depends on the Server configuration information. The role of these CGI scripts in the Web interfaces is discussed in Section 7.1. None of these scripts should be executed manually, as most are called internally by the Server within a specific shell environment or by the Web server with specially formatted arguments or input.

5.2 Submission Flow

To explain the flow of submissions through the NEOS Server, we begin with a holistic approach that briefly mentions the main components in Table 5.2 as part of the process. We then move on to a more detailed examination of the various scripts that the Server comprises.

Jobs are processed by the Server in pipeline fashion, beginning with the script *receiver.pl* and ending with the script *job_end.pl*. The submission flow becomes somewhat complicated by the parallel execution of certain scripts like *parser.pl*, *parse.pl* and by the *job* scripts. Figure 5.2 shows two simultaneously arriving jobs: one from the Web and one as an email. After being sent to the NEOS Server via any submission interface, the job submissions are located by the *receiver*. When it sees submissions, the *receiver* calls the *initializer.pl* script to add the submissions to the NEOS jobs directories. Then the *initializer* spawns *parsers* to decode the submissions. When the *parsers* complete their tasks, the *scheduler* is informed of the waiting jobs. The *scheduler* makes every effort to refer the jobs to the correct solver in the order they finished being parsed and spawns a *job_start.pl* script for each job in turn. The *job_start* scripts request job execution on the solver station determined by the *scheduler* via a *job_client* script written to handle socket communications with the remote NEOS Comms daemons. When the job is complete, the remote communications daemon sends results to the Server through the *socket-server*, which calls a *job_end* script to coordinate returning results to the user.

While the Web and socket interfaces display a similar flow of information, Email submissions require different amounts of attention at various stages. For example, Figure 5.2 depicts how Web submissions are presented to the *receiver* script by NEOS helper scripts (CGI scripts). On the other hand, the *receiver* must actively request Email submissions from the email server. The *parser* emails users that their Email interface submissions have arrived, and the *job_end.pl* script emails results back to the user through an external mail program (MH) rather than relying on NEOS helper scripts. The *job* scripts make no effort to transmit intermediate results to the email user as they do to the helper scripts; instead they wait until the job is complete to send results.

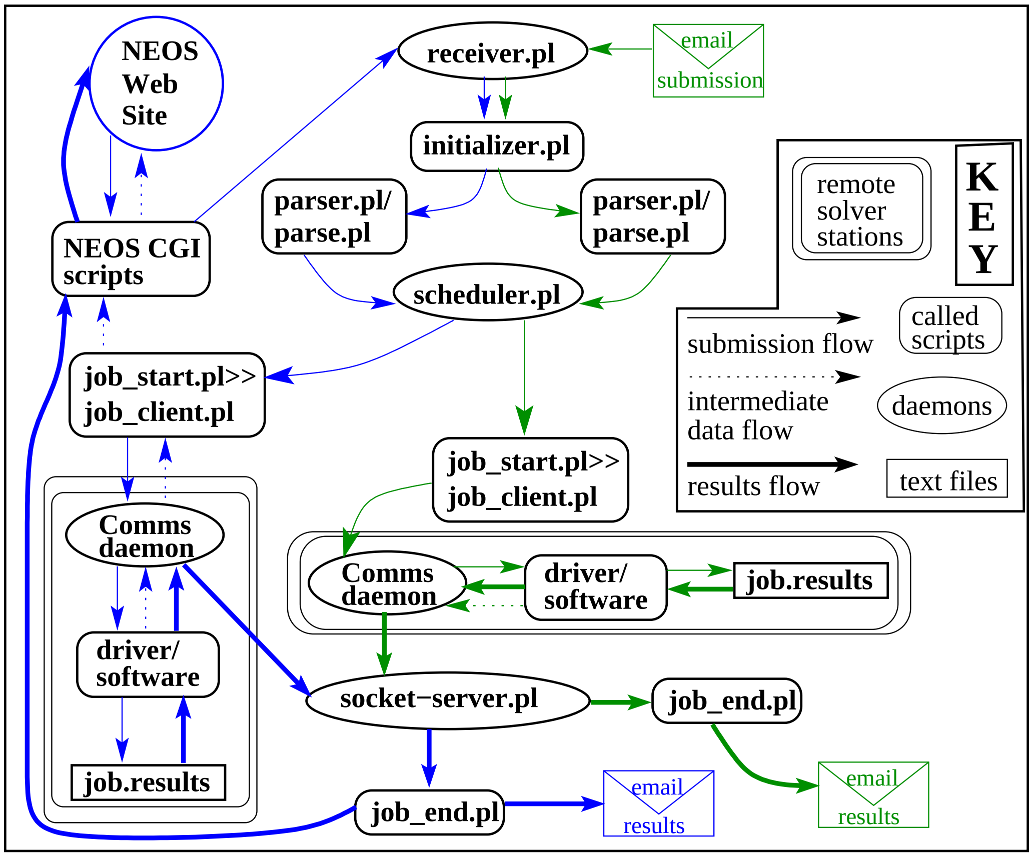


Figure 5.2: NEOS Server Submission Flow Example

One distinction between the Web and socket submission interfaces is that the NEOS-created *socket-server* daemon for writing NST submissions runs on the NEOS Server machine while the Web server daemon mentioned is the Web server (e.g., Apache) that hosts the NEOS Web site. The Web interface requires CGI scripts to facilitate communication with the NEOS Server via files while the *socket-server* written for NEOS contains knowledge of the NEOS Server protocols internally.

Once we have looked at the components of the job submission flow that are common to all interfaces, we examine the more popular interfaces in some detail. Section 5.3 deals with the *socket-server* both as a user interface and an interface between the NEOS Server and the Comms Tool daemons. The CGI interfaces to the Server are discussed as a part of Section 7. The following descriptions of Server scripts are common to all user interfaces except where otherwise noted.

5.2.1 The Receiver

The script *receiver.pl* is one of the three Server daemon processes that should always be running, and the job of the *receiver* is to check for incoming jobs from any one of the interfaces in its spooling directory.

NST jobs are initially written to the directory *server_var*/spool/SOCKET, Kestrel jobs are found in *server_var*/spool/KESTREL, and, similarly, WWW submissions are deposited in the directory *server_var*/spool/WEB. When one of the NEOS Server interfaces is writing a job submission file to its spooling directory, it must first choose a unique file name, which takes the general form

Table 5.2: **Server Scripts & Executables**

Executable	Main Purpose
receiver.pl	looks for incoming jobs and invokes initializer
initializer.pl	creates job directories & invokes parser.pl
parser.pl	parses jobs into files via parse.pl; signals scheduler.pl
parse.pl	performs the actual parsing using token config. files
address	extracts email address from all submissions
scheduler.pl	handles job queues; forks job_start.pl's
counters.pl	solver usage counts/restrictions & size limit
client.pl	code for generic socket requests
job_client.pl	code for begin job requests
job_start.pl	requests Comms daemon execute job
job_end.pl	returns results (or signals they are ready)
socket-server.pl	handles socket requests to Server

interface.unique_filename. In order to ensure that all job files are completely written before being collected by the *initializer*, the *socket-server*, *nph-solver-*.cgi* scripts, and the Kestrel server will touch the file DONE.*interface.unique_filename* when finished writing the submission file and then signal a job arrival by touching the file MAIL in the interface's spooling directory. The original NEOS Server implementation checked for Email jobs in much the same way as other jobs by looking for a nonempty /var/spool/mail/*username* file. Since checking mail via the network file system has fallen out of favor, the current *receiver* script queries the POP3 mail server for new mail.

The *receiver* checks for MAIL files every second and queries the mail server only after a number of unsuccessful rounds through the spooling directories so as not to overwhelm the mail server. It launches the *initializer* as a blocking call if there are submissions waiting, with the argument `-mail` if there is new mail waiting on the mail server.

5.2.2 The Initializer

The script *initializer.pl* is invoked by the *receiver* once jobs have arrived from any one of the interfaces. The primary job of the initializer is then to

- download email from the mail server, and refile messages in the *server_var*/tmp directory if the `-mail` argument is present;
- throw out any junk mail;
- assign a job number to each legitimate submission;
- create the job directory *server_var*/jobs/job.*number*, where *number* is the number assigned to the job by the *initializer*;
- move the job from its spooling or temporary directory (i.e., *server_var*/tmp, *server_var*/spool/SOCKET, *server_var*/spool/WEB, or *server_var*/spool/KESTREL) to the file name job.received in its new job directory; and
- fork the script *parser.pl* to parse the contents of job.received.

The *initializer*'s first order of duty in each spooling directory involves checking for and deleting the MAIL file. If MAIL is present, the script looks for a matching DONE file for every submission and ignores those files lacking a match. The *initializer* assigns a job number based on the contents of *server_var*/lib/next_serial_number, which it increments, and outputs a file for later input by the Web, Kestrel, and socket servers. The job number written to this file serves as a handle to these interfaces to obtain intermediate output from a solver as well as final job results.

While users of the Email interface do not normally receive intermediate results, once the *parser* has found a return address, it sends reply mail with the job number and password so that users can check the status of their jobs on the Web or refer to the job number with questions about particular submissions.

5.2.3 The Parser

Parallel execution in the Server begins with the script *parser.pl*, as the initializer forks a new Perl process for each job. The primary role of the *parser* is to read in the file job.received and decompose the job into its constituent parts:

- Address of the sender
- Solver requested
- Data to be input to the solver

Once the sender's IP or email address has been determined (via the *address* executable examining the "`From:`" line in job.received), this information is saved in the file job.address. Because both socket and WWW submissions will yield i.p. addresses, the WWW submission addresses are tagged with `WEB_USER`; and the addresses of sample WWW submissions have `TRIAL_WEB_USER` prepended to them. The solver requested is saved in the file job.type. The *parser.pl* script then calls on *parse.pl* to compare the data in the remainder of job.received with the information in the requested solver's token configuration file and map the data to files accordingly. The NEOS Server interfaces that token delimit files for the user can make parsing somewhat simpler by omitting the end token and giving the begin token as `begin.token[`*size*`]`, where size is the number of bytes of content for the file, beginning on the next line. The *parse.pl* executable also checks whether any of the files submitted were in compressed format and, if possible, decompresses them. The *parse.pl* script further massages the data so that input from DOS or Mac operating systems resembles that expected by a Unix-based solver. The *parse.pl* script excludes binary files, which are flagged by a begin token ending in *BINARY*, from this operating system format conversion. The correct parsing of binary files relies on having the *size* available.

When *parse.pl* has finished its tasks, the parser is ready to request scheduling of the job for execution. The *scheduler* has its own spooling directory in *server_var*/spool/ (similar to those checked by the *receiver*). To wake up the *scheduler* daemon, the *parser* saves the job type into the file job.*number* in the directory *server_var*/spool/JOBS/ and signals the *scheduler* by creating the empty file *server_var*/spool/JOBS/JOBS.

5.2.4 The Scheduler

The *scheduler.pl* script is the second of the three Server daemons, and its primary goal is to fork a *job_start.pl* script for each job. The *job_start* script is then responsible for connecting to the NEOS Comms Tool daemon running on the machine of the remote solver. Jobs that cannot be scheduled on their requested solver are also handled by the *job_start* script on the NEOS Server host machine, which will generally return job results explaining why the job could not be scheduled.

The important information needed by the *scheduler* for each job is the job number, the job type and solver identifier, the machines (and port numbers) running the solver's NEOS Comms Tool daemon, and the jobs already executing on each solver station. The *scheduler* internally stores a list of all job numbers, their solver, and their state (e.g., executing, queued for execution, scheduling error). The *scheduler* also tracks the current station use for each solver (e.g., solver GAMS:BDMLP has 3 jobs running on its fire.mcs.anl.gov station).

The *scheduler* executes only as many jobs of one type on a solver station as the solver administrator indicates are allowed on that station when registering the solver. The *scheduler* also contains a hard limit on NEOS jobs that may be scheduled on a machine, regardless of the solver accessed. The limit should be adjusted if it is not appropriate for the type of machines being used (for example, the solver station actually serves as a scheduler for a large number of machines or processors where the real work is done). The machines that are currently executing jobs are represented by the files in the directory *server_var*/proc/stations, where each file represents a solver on a particular machine. For example, the file fire.mcs.anl.gov-GAMS:BDMLP is created by the *scheduler* to show that the GAMS:BDMLP solver is currently running on the machine fire.mcs.anl.gov. If the file is nonempty, it will contain an integer representing the number of GAMS:BDMLP jobs running on fire. It is the responsibility of the *scheduler* to decrement or remove this file when the forked *job_start* script has completed execution.

The file *server_var*/lib/station_type_port is maintained by the *scheduler* and stores a list of all solver machines, including which port the NEOS Communication Packages are listening on. The *scheduler* will consult this list before forking a *job_start* script in order to direct it to a particular machine and port. For example, the machine/port entries in *server_var*/lib/station_type_port for the GAMS:BDMLP solver might look like

```
fire.mcs.anl.gov:GAMS:BDMLP:4001
lava.mcs.anl.gov:GAMS:BDMLP:4000
ember.iems.northwestern.edu:GAMS:BDMLP:4007,
```

which gives the *scheduler* three machines to choose from when executing a job.

5.2.5 The Job Scripts

The *job_start.pl* script is responsible for initiating the data exchange between the NEOS Server and remote solver stations for all job submissions. In the event that a job cannot be scheduled on a remote machine, the *job_start* script writes a message to the results file and mails users of the Email interface. The *job_start* script requests that legitimate job submissions be executed on their solver stations. If the solver station is *localhost*, as is the case for the ADMIN solvers, *job_start.pl* finds and executes the solver driver itself, as discussed in Section 6. In order to execute jobs on remote machines, the NEOS Server must act as the client making requests to the NEOS Comms Tool daemon as the server. For these submissions *job_start.pl* executes the *job_client.pl* script, a specialized version of the *client.pl* script distributed with the Comms Tools and the NST.

The *job_client* actually connects to the NEOS Comms Tool daemon on the remote machine and writes job intermediate output as it is streamed back. Because the remote Comms Tool daemon contacts the NEOS *socket-server* with job results, the *socket-server* bears the onus of calling a *job_end* script that logs the job in the *server_var*/databases/master, signals the user interfaces that collect results themselves that results are available, and flags the job as terminated for the benefit of the *scheduler* and *cleaner*. In theory, the *job_client* script should be able to handle job termination because it receives a special token from the Comms Tool daemon when job results have been sent to the Server. In practice, the *job_client* script is the longest running Server script associated with a job and therefore faces the highest probability of accidental death. Relying instead on the *socket-server* to initiate the call to *job_end* enables cleanly returned results from the remote station to be passed on to the user.

Of course, the Comms Tool daemon on the remote machine may fail unexpectedly in ways that prevent it from closing its socket connection with the *job_client*. The *job_client* therefore utilizes connectivity checks available through the Internet Protocol to determine failure of the Comms Tool daemon to maintain its socket connection. When the *job_client* has flagged a job's Comms Tool connection as prematurely dead, the *cleaner* will attempt to connect to the Comms Tool daemon to retrieve any job results on the remote system. The call can also determine whether the job is still running on the solver station. It may seem bizarre to appoint this task to the *cleaner*, which may not run too often, but transferring the responsibility to a single script that runs occasionally allows the *job_client* for any crippled jobs to exit. Also, immediate detection of abnormalities by the *job_client* is an illusion. Operating system–dependent implementations of the IP standard tend to wait hours between socket checks. The Comms Tool daemon may have contacted the *socket-server* and returned job results long before the *job_client* notices its socket connection has gone bad.

The *job_start* script relies on *job_client.pl* to pass on requests to the NEOS Comms Tool daemon to accept uploaded user data and begin execution of the remote solver driver. The *job_start* script executes the *job_client* script with all of its output redirected to the file job.out so that the standard output from the solver driver being piped by the NEOS Comms Tool daemon and any error messages from the NEOS Comms Tool daemon can be displayed to the user by the socket and Web servers. When the job is complete, the *job_end* script forwards results to Email users and WWW interface users who request email.

The *job* scripts also keep the master database. When results are in, *job_end.pl* writes a 256-byte descriptive entry into the master database at the position in the database corresponding to the job number. Then *job_end.pl* signals the waiting WWW and socket servers that the job is finished by writing an end token to the job.out file, and it creates the empty file DONE in the job's working directory to mark the job as completed. In the case of administrative jobs that are not sent to a remote solver station, *job_start.pl* also handles these termination tasks. Because job completions will not always occur in the order in which the jobs are received, the *job_end.pl* will fill in blank entries with spaces until the master database is of the correct length for its job entry. If the space is not filled, the database becomes corrupted by areas of binary junk. The *report* and *cleaner* scripts still contain code to replace any unreadable sections of the master database with white space so that the Server administrator can query the database as part of Server monitoring.

5.3 The Socket Server

The *socket-server* script is responsible for responding to all requests from the socket submission interfaces and solvers (via the NEOS Comms Tool daemon).

In general, a TCP/IP server listens for and accepts socket connections with arbitrary clients. After being connected, the *socket-server* reads the client's request from the socket, processes the request, and sends a response back over the same socket connection. The Perl TCP/IP client server/bin/*client.pl* is distributed with the socket NSTs and the Comms Tool to make requests to the NEOS Server that the *socket-server.pl* script can fulfill.

5.3.1 Interface with the Job Pipeline

When the *socket-server* receives a job from a client, it must submit this job to the Server on behalf of the client. The entry point for these jobs is the spooling directory *server_var*/spool/SOCKET. In this directory the *socket-server* writes job submissions with a header. In other words, all submissions look similar to email submissions in that they start with a "From:" line. The NST at the user end actually composes the job submission, and the *socket-server* just reads in the data over the TCP/IP connection. Then the *socket-server* creates a file in *server_var*/spool/SOCKET under the name job.*number*, where *number* is an internal variable in the *socket-server* used to track incoming jobs.

After job.123, for example, has been written, the *socket-server* signals the *receiver* by creating the empty file *server_var*/spool/SOCKET/MAIL. At this point the *receiver* wakes up, and the job is processed by the *initializer*.

The *socket-server* will know that its job has been accepted once the file job.123 has been renamed to job.123.number. This new file will contain the job number assigned by the *initializer*. This job number will need to be returned to the client that originally submitted the job because it will be used as a handle when later downloading the job results.

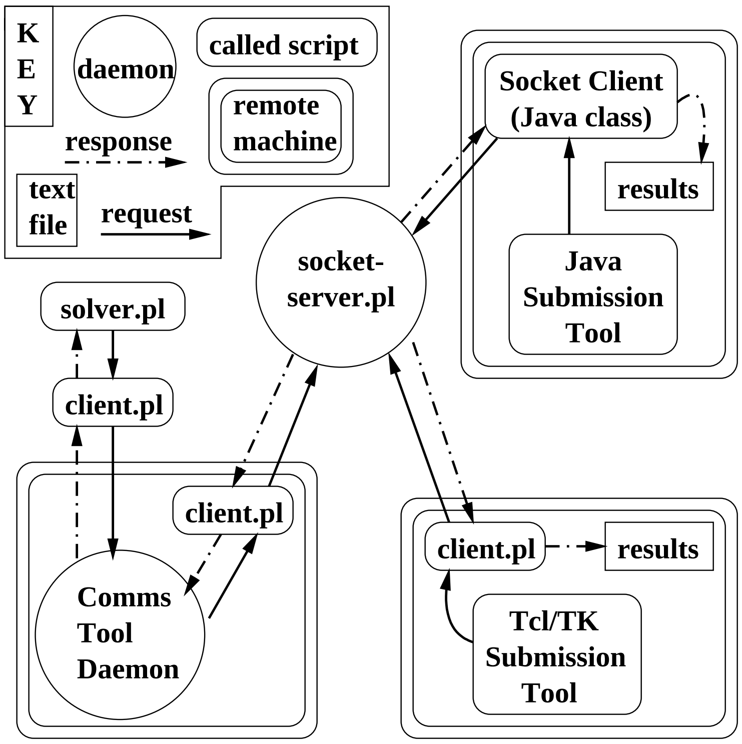


Figure 5.3: NEOS Socket Communication

5.3.2 Socket Communications Requests

The sections below discuss the requests accepted by the *socket-server*, how it processes each request, and the response sent back over the socket to the client. There are two categories of requests: those issued by the *client* script or Java *SocketClient* on behalf of an NST and those issued by *client.pl* on behalf of a Comms Tool daemon.

The first line of each request to the *socket-server* represents the userid of the client, and the second line represents the actual request. After the userid has been read by the *socket-server*, the machine name is determined (as per the TCP/IP protocol), and the request is then read. The userid and machine name are used for logging purposes so that the logs display all requests to the socket server.

All requests can be tested by executing *client.pl* directly using the `perl` command with the arguments

```
client.pl -server <hostname>:<port> -request <server request> ,
```

where `<hostname>` represents the machine name of the server (e.g., `opt.mcs.anl.gov`), `<port>` represents the port the *socket-server* is running on (e.g., `3333`), and `<server request>` represents the actual request (e.g., `verify`). The *client.pl* script is coded to automatically send the userid of the requester.

Unless otherwise stated, the first line of server output in response to a request represents the amount of data to be transferred back to the *client*. All lines thereafter represent the output of the request.

5.3.3 NEOS Submission Tool Requests

The requests in this section describe the protocol between the NEOS *socket-server* and the NSTs.

solver_list

The request `solver_list` returns to the client a list of all the registered solvers. Each line of output contains information of the form

Solver Name=TYPE:SOLVERID

The following line of output, for example, would be transferred for the *TRON (AMPL input)* solver whose type and id is BCO:TRON-AMPL:

```
TRON (AMPL input)=BCO:TRON-AMPL
```

admin_list

The request `admin_list` returns to the client a list of all administrative solvers in addition to all registered solvers. The output of this request is identical to that generated by `solver_list` except that the administrative solvers are included, of which there are currently only three:

```
Adding/Modifying a Solver=ADMIN:ADDSOLVER
Enabling/Disabling a Solver=ADMIN:STATUS
Kill Job=ADMIN:KILL_JOB
```

helpTYPE:SOLVERID

The request `help`TYPE:SOLVERID returns to the client the NST help file for the requested solver.

configTYPE:SOLVERID

The request `config`TYPE:SOLVERID returns to the client the token configuration file for the requested solver.

begin job *size*

The request `begin job` *size* is used to submit jobs to the *socket-server*, where *size* is the size in bytes of the job being submitted. After the job has been submitted, the first line of output to the client is the job number, and all remaining lines represents the standard output/error from the executing job. These lines are transmitted as they are generated by the solver. There is no line specifying the size of the transfer from the *socket-server*, as this cannot be predetermined while the solver is executing.

The following example can be followed to submit a job to the *socket-server* specifying `help` for the administrative solver ADMIN:ADDSOLVER.

First, the command

```
perl server/bin/client.pl -server opt.mcs.anl.gov:3333 \
      -request "begin job 33"
```

is executed.

After the connection has been established, the output should be similar to

```
client.pl: connecting to opt.mcs.anl.gov:3333
client.pl: connected
client.pl: sending request
```

The following lines are typed as input to the executing script, followed by an end signal, usually CTRL-D:

```
type admin
solver addsolver
help
```

The remaining output from the server should be similar to

```
client.pl: 33 bytes sent
client.pl: receiving data
7773
Welcome to NEOS!
<CLEAR_SCREEN>
Parsing:
        0 bytes written to help.type ()
Welcome to NEOS!
<CLEAR_SCREEN>
Scheduling:
  You are job #7773.
  Solver Queues:
    ADMIN:HELP: 7773:
  Jobs Executing:
    No jobs executing.
<CLEAR_SCREEN>
Parsing:
               <END_STANDARD_OUT>
        0 bytes written to help.type ()
client.pl: 339 bytes received
client.pl: exiting
```

For now, all of the marked-up commands, such as <CLEAR_SCREEN>, can be ignored, as these are directives to the NSTs and Web CGI scripts to do exactly that, clear the screen for the user. What is important to note is the job number that was returned as the first line of output, 7773 in this example. Following this job number is the standard output/error from the running job. This is the same information that is written into the file job.out in the job's temporary working directory in *server_var*/jobs.

get results *number*

After a job has completed, the NST retrieves the results using the request `get results` *number*, where *number* is the number of the job and should have been returned as the first line of output when the job was submitted with the request `begin job`.

In the preceding section we were assigned a job number of 7773. The *client.pl* can send this request to the *socket-server*:

```
perl server/bin/client.pl -server opt.mcs.anl.gov:3333  \
      -file myResults -request "get results 7773"
```

The output should be similar to the following:

```
client.pl: connecting to opt.mcs.anl.gov:3333
client.pl: connected
client.pl: sending request
client.pl: 17 bytes sent
client.pl: receiving data....
client.pl: 26348 bytes received
client.pl: exiting
```

To see the actual job results, one must view the file myResults. Without the `-file` option the results would have been displayed directly to the console.

submit-*version*.tk

The request `submit-`*version*`.tk` is used to retrieve the Tcl/Tk code for the Unix NST. This is the code responsible for generating the NST GUI. The version number is used to distinguish between different versions in the event that future Tcl/Tk NST distributions require different GUIs. Currently, the only version of the Tcl/TK code is `submit-1.0.tk`.

java-client-*version*.jar

The request `java-client-`*version*`.jar` returns the Java archive file including all of the java Submit Client class files. If the *version* requested is the same as the most recent version of the Submit Client, then the user's version is already up to date; and the *socket-server* returns a message to that effect. Otherwise, the latest version is returned.

submit-main.txt

The request `submit-main.txt` returns to the client the "homepage" for the NEOS Submission Tools. This text is displayed in the main window of the NST when it is first started.

submit-help.txt

The request `submit-help.txt` returns to the client the help file for the NEOS Submission Tools. This text is displayed in the main window of the NST when the Help button is clicked.

verify

The request `verify` may be used by NEOS Submission Tools to verify the identity of the NEOS Server. The output of this request is the name of the Server.

5.3.4 Comms Tool Requests

The requests discussed in this section are used by the Comms Tool daemons to coordinate all communication between a solver and the Server. The primary responsibilities of the Comms Tool are to accept connections from the *socket-server*, download user data, execute the solver, and return the job results.

Just as with the Tcl/TK NST, the Perl script *client.pl* is distributed with the Comms Tool and is used to issue all requests to the *socket-server*.

verify

The request `verify` may be used by Comms Tool to verify the identify of the Server. The output of this request is the name of the Server.

begin results *size number*

The request `begin results` *size number* is used to return job results to the Server. The arguments represent the size of the job results in bytes and the job number.

comms-*version*.tk

The request `comms-`*version*`.tk` is similar to `submit-`*version*`.tk` in that it downloads the Tcl/Tk code used to create the Comms Tool GUI. The version number is used to distinguish between different version in the event that later distributions of the Comms Tool require a different GUI. Currently, the only version is 1.0.

comms-daemon-*version*.pl

The request `comms-daemon-`*version*`.pl` returns the Unix Perl code for the main communications daemon that accepts incoming connections from the server. Currently, the only version is 1.0.

comms-backup-*version*.pl

The request `comms-backup-`*version*`.pl` returns the Perl code responsible for editing the crontab file in the event that the crontab entry for the communications daemon needs to be enabled. Currently, the only version is 1.0.

disable-backup-*version*.pl

The request `disable-backup-`*version*`.pl` returns the perl code responsible for editing the crontab file in the event that the crontab entry for the communications daemon needs to be removed. Currently, the only version is 1.0.

register *solver password port*

The request `register` *solver password port* is issued by *client.pl* on behalf of a solver's communication daemon. This request tells the NEOS Server what port the communications daemon is listening on. This request is sent to the *socket-server* whenever a communications daemon is restarted.

deport *solver password station port*

The request `deport` asks that a registered solver have its entry in the *server_var*/lib/station_type_port file for this station and port removed. This request is issued when the solver administrator *disables* a solver in the Comms Tool GUI and ensures that no submissions for that solver are sent to that port. This is useful to disable a particular solver on a master Comms Tool daemon.

5.3.5 Server Requests to the Comms Tool

This section discusses the requests accepted from the NEOS Server by the Comms Tool daemon (*comms-daemon-version.pl*).

verify

The request `verify` is made by the Server to verify the identity of a communications daemon. The output returned to the Server from the communications daemon is of the general form TYPE:SOLVERID`@hostname:port`, where the type and solver identifier combination ALL:ALL is recognized as being a single master daemon for handling requests to any of a group of enabled solvers.

begin job size number

The request `begin job` *size number* is made by the Server to remotely execute a particular solver, where *size* represents the size of the user data to be transferred to the Comms Tool and *number* represents the job number. This job number is later used in the `begin results` request after the application exits and the Comms Tool daemon returns the results to the Server.

The user data uploaded to the Comms Tool represents the contents of a file created with `tar` and compressed with `gzip`. The Comms Tool must uncompress and untar the data in order to present the user's data files to the solver. When the solver driver is executed, the data has already been untarred and uncompressed so that the files are in the current working directory.

check job number

The `check job` *number* request may be made by the NEOS Server *cleaner* script when it happens upon a job directory that contains the flag files indicating the socket connection with the remote solver station has been broken but that does not have the flags to show the job has been terminated properly. The Comms Tool daemon that receives this request will check its .comms directories for the process identifiers associated with the job. Depending on the state of those processes, it may wait for a job whose Comms process has terminated to finish and then return results, it may return results immediately for a job that has already finished, or it may return a message saying that it failed to find any results for a dead job. The other possibility is that an older version of the Comms Tool daemon may not recognize the request. The *cleaner* should handle any of these cases.

kill job number

The request `kill job` *number* may be made by the NEOS Server through the ADMIN:KILL_JOB solver. If the *comms-daemon* script running on the remote solver station's list of running jobs includes the *number* requested and the solver's jobs have been labeled "killable" by the solver administrator when starting the Comms Tool daemon, then the *comms-daemon* will kill the entire process group associated with that job. Solver administrators should test the ADMIN:KILL_JOB solver's behavior with their solver to make sure that killing the process group does effectively kill all descendants of the child job. Some Unix systems handle process groups differently from others. Also, some application software may reset the process group id for its own ends or run some processes on other machines, so ADMIN:KILL_JOB may not work for all solvers. In these cases, the Comms Tool daemons for the solver should simply be started with the "killable" option deselected.

6 Administrative Solver Implementation

Four solvers currently come packaged with the NEOS Server 4.0. These solvers, introduced in Section 4.1, are identified as ADDSOLVER, HELP, KILL_JOB, and STATUS. The registration information for these solvers can be found under the directories server/lib/admin_solvers/ADMIN:*identifier*/, stored in the same format the ADMIN:ADDSOLVER uses for other solvers. The source code for the administrative solvers is also present in these directories so that it can be executed directly on the NEOS Server host machine. Because they are integrated with the Server and run under the Server userid, ADMIN solvers may call on Server scripts to complete their tasks or write to Server directories.

The submission interfaces for the administrative solvers are created by *config.pl*. To allow for customization, the *config* script makes substitutions or additions to any files with the name of a standard solver registration file followed by a .server suffix. The amended file is then renamed to the corresponding registration file name. Because their registration information is stored similarly, the interfaces of the administrative solvers are constructed in the same way as for other solvers, allowing the same familiarity afforded by the automated addition of solvers to the NEOS Server pool.

When a job is submitted to an administrative solver, the submission arrives on the Server in the appropriate *server_var*/spool/ directory according to the interface of origin. The *parser* script is the first part of the submission flow that distinguishes an ADMIN job from other jobs. The parser allows users greater flexibility in invoking the ADMIN:HELP solver by sending any submission with *help* in the first nonblank line after the header to solver ADMIN:HELP, regardless of other parsing oddities. The *scheduler* also makes special dispensation for ADMIN jobs by automatically assigning them *localhost* as their solver station and by redirecting the search for their solver registration information to the directory server/lib/admin_solvers/. The *job_start.pl* script assigned to an ADMIN job makes no effort to contact a Comms Tool daemon through the *job_client.pl* script. Instead, *job_start.pl* reads the name of the driver for each ADMIN solver from ADMIN:*identifier*/SOLVE, invokes the administrative solver driver, and undertakes job termination tasks itself. With these relatively minor adjustments, ADMIN jobs can be submitted and processed like any others.

The *Adding/Modifying a Solver* administrative solver comprises a group of Perl scripts for checking registration information and adding solvers to the NEOS Server. If each piece of data sent in by solver administrators is in the correct format, the ADDSOLVER ensures that there is a directory for the solver under *server_var*/lib/solvers/. In the case of a registration update, old solver stations in the file *server_var*/lib/solvers/*type*:*identifier*/STATIONS not in the new list are slated for deletion by the *scheduler* from its list of available solver stations and the ports bound by their Comms Tool daemons. The ADDSOLVER copies the new list of stations and other information to the correct *server_var*/lib/solvers/*type*:*identifier*/ directory. The ADDSOLVER adds or updates the solver's entry in the *server_var*/lib/solver_list. Then it calls server/bin/*make-solver.pl* with the correct argument to build the Web pages and CGI scripts for the Web interface to the solver. When the process is finished, solvers are registered as available to all interfaces enabled for the Server.

The *Enabling/Disabling a Solver* administrative solver, STATUS, changes the availability of a registered solver. Depending on the data sent in with a STATUS job submission, STATUS may just disable a solver by removing it from the *server_var*/lib/solver_list and calling on server/bin/*make-server-solvers.pl* to update the Web page listing of solvers. The reverse operations may be performed at any time to enable such a solver again. If the solver administrator requests deletion, the solver is first disabled. Then the solver registration information is deleted from under *server_var*/lib/solvers/, the Web pages and scripts are deleted, and the solver is removed from the *scheduler*'s list of stations and ports. The solver can be restored only by reregistering it through ADMIN:ADDSOLVER.

The *Help* solver can give help relating to a particular solver or for the NEOS Server in general. The *Help* solver provides individual solver help aimed at the Email interface only. The solver help files for the Web interface are built into the Web site, and the solver NST help files are available by request to the *socket-server*. The ADMIN:HELP solver searches the library directory of the solver requested for the email-help file. If present, HELP returns this text to the user along with information on which interfaces are currently enabled for the NEOS Server. If the solver administrator fails to provide an email help file, HELP informs the user that no help is available. If the user does not request a particular solver or the help submission cannot be parsed, *Help* returns a general_help text from its own directory along with a list of all registered solvers, including administrative solvers. It also provides further tips on how to use the ADMIN:HELP facility.

The *Kill Job* administrative solver relates to specific jobs. If the requested job number's directory still exists under *server_var*/jobs/ and ADMIN:KILL_JOB can find the job's solver station and port, then *Kill Job* sends a socket request through the Server's server/bin/*client.pl* script to the Comms Tool daemon associated with that solver and station. *Help* returns the output from the request to the user, which usually states that the job is dead, the KILL_JOB solver is not enabled by that Comms Tool daemon, or that the daemon has no record of the job. The correct implementation of the KILL_JOB request by the Comms Tool daemon is hence not required, because it would be impossible on some systems. Solver administrators are expected to test the feature on their site before enabling it. The administrative *Kill Job* solver performs an attempt at its task, regardless.

As we contemplate new features for the NEOS Server, we may decide to implement new administrative solvers. The task is not too difficult for any Server administrator. If the format of the existing administrative solvers is followed and the new solver added to the server/lib/admin_list, then the administrative solver becomes available with a simple rebuild of the Server.

7 Web Site/Interface Implementation

If a Web site is not desired or cannot be supported, the NEOS Server will still function fully via its socket and Email interfaces. We recommend a Web presence for advertising and educational purposes, however, and note that, for the NEOS Server for Optimization, this is our most popular job submission interface.

Many of the CGI scripts in Table 7.3 support miscellaneous user requests that can be made throughout the Web site. Whenever a user clicks on a NEOS Web form's `Submit` button, the Web server that hosts the page executes a NEOS CGI script. For example, when users send in comments from either the NEOS Server comments page or one of the individual solver comments pages, *server-comment.cgi* or *solver-comment.cgi* handles emailing the comments to the appropriate addresses. When users submit their email address for the *neos-news* list, *list.cgi* forwards their request to the list server. The addresses of people downloading various NEOS packages are recorded by the *download* scripts. The other files in Table 7.3 play a more active role in the job submission flow as it relates specifically to the Web interfaces.

Table 7.3: **Scripts and Executables for the Web Server**

General Template	Main Purpose
cgi-lib.pl	parses Web uploads; sets upload limit
check-pwd.cgi	verifies check-status password; outputs job results to Web browser
check-status.cgi	acts as a gateway to job results
downloads.cgi	logs emails of Tcl/Tk tool downloaders
java-downloads.cgi	logs emails of Java tool downloaders
list.cgi	adds email addresses to neos-news mailing list
server-comment.cgi	emails user comments to Server administrator
server-downloads.cgi	logs emails of NEOS-4.0 downloaders
tempfile.cgi	creates temporary, unique files names

Solver Script Template	Main Purpose
nph-solver-sample.cgi	handles sample submissions
nph-solver-www.cgi	handles Web interface submissions
solver-comment.cgi	emails user comments to solver administrator

7.1 The Web Server Role in Submission Flow

The NEOS Server acts to take full advantage of Internet technology by creating a potentially extensive Web site. In addition to the Server homepage, solver listing, and various informational pages, the NEOS Server creates a homepage for each solver added. From these solver homepages, users can access the NEOS Server's Web interfaces for job submission.

The NEOS Server works hand in hand with a Web server to orchestrate Web interface submissions. The *nph-solver-www.cgi* script, in conjunction with the *cgi-lib.pl* routines, uploads a user's files and other data entries. By referring to the solver's token configuration file, it composes one submission file in the NEOS Server's token delimited format. Once the submission file has been created and given a unique file name by *tempfile.cgi*, *nph-solver-www.cgi* leaves a signal of the pending submission for the *receiver* in the form of an empty MAIL file under *server_var*/spool/WEB/. The *initializer* facilitates communication for Web interface users by writing the job number assigned to the submission to a file the *nph-solver-www.cgi* script expects to find. Once *nph-solver-www.cgi* has the job number, it can read in the intermediate information being written to *server_var*/job/job.*number*/job.out by *job_client.pl* and write it in server-push HTML format. The tag, `<END_STANDARD_OUT>`, that *job_end.pl* writes to job.out tells *nph-solver-www.cgi* that the job results are available. CGI script *nph-solver-www.cgi* then formats the results as HTML text and prints a redirection tag from the intermediate output to the final results.

The solver's `Sample Submission` interfaces create output through the *nph-solver-sample.cgi* script similar to what users would see from the *nph-solver-www.cgi* script, but *nph-solver-sample.cgi* expects sample submissions in token-delimited format. Instead of uploading user files and inserting the solver tokens, *nph-solver-sample.cgi* downloads the selected Web-accessible sample file registered by the solver administrator from its URL and signals the Server of a waiting Web submission through the MAIL file. The CGI scripts' method of interacting with the NEOS Server through files necessitates that the Web server have access to the file system where the NEOS Server's *server_var* directory is mounted. The two servers need not reside on the same machine, but they must share certain files to work together.

From the user perspective, the Web forms offer a straightforward, probably familiar, interface. The users submit information via a Web form, watch the intermediate information about the solution stages grow as the Web server sends available data, and finally view the job's results. Because pushing results from a Web server to a browser over an extremely lengthy time (e.g., when a job runs for hours) is prone to network failure, users are given job number and password information at the top of the intermediate results page as well as the URL of the *check-status.cgi* script. Through the Web page created by *check-status.cgi*, users are able to submit their job number and password and request the latest intermediate results or the job's final results. The *check-pwd.cgi* script verifies the job password and retrieves text directly from the *server_var*/jobs/job.*number* files. Because *check-pwd.cgi* finds its information from the job directory, users who have sent in their submissions to the Server through other interfaces can also retrieve job information through the Web.

7.2 Web Site Creation

The NEOS Server builds its core Web site using the configuration information provided by the Server administrator. The NEOS Server requires such critical information as the name of a Web-accessible parent directory where it can write its HTML files and a similar location for CGI scripts that can be executed by the Web server. The Server administrator also needs to know the URLs for these directories before NEOS can build its site so that the site can be fully interconnected. The scripts responsible for creating the Web site reside under server/bin/, and their names generally begin with *make*, as shown in Table 7.4. Once the Server configuration script *config.pl* has collected all of the necessary information during the interactive `make` process described in Section 2.1, *config.pl* calls on *make-all.pl* to build the Web site.

The script *make-all.pl* contains a routine for taking the template HTML files under server/lib/html/ and replacing certain tags with Server configuration information. Most of the pages to which the Server homepage links directly are built by *make-all.pl* along with their associated CGI scripts as necessary (for example, the *list.cgi* script to add people to the *neos-news* mailing list). Completed HTML files are written under the parent Web directory supplied by the Server administrator, and completed CGI scripts are written to the CGI directory supplied to *config.pl*.

The *make-all.pl* script then calls other *make* scripts to build the more complicated remainder of the Server Web site. The *make-server-homepage.pl* script builds the Server homepage from two files under server/lib/html/. The *make-server-solvers.pl* script creates the page listing available solvers by reading in registered solvers from the *server_var*/lib/solver_list and server/lib/admin_list. The general Server comments Web page and CGI script are created by *make-server-comments.pl*. Then *make-all.pl* calls on *make-solver.pl* to build Web homes for solvers that the Server already has

Table 7.4: **Scripts to Generate HTML & CGI**

"make-" Prefixed Script	Main Purpose
all.pl	builds misc. pages; calls other make-'s
server-homepage.pl	makes server's homepage
server-solvers.pl	makes solvers Web page
server-solver-in.pl	makes alternative solvers Web page
solver.pl	calls other make-solver-* scripts
solver-comment.pl	makes solver's comment form
solver-comment-cgi.pl	completes solver's comment CGI script
solver-email.pl	completes solver's template HTML page
solver-homepage.pl	makes solver's homepage on server
solver-sample.pl	builds solver's sample submission form
solver-sample-cgi.pl	completes sample submission CGI script
solver-template.pl	builds solver's email template
solver-www.pl	builds solver's Web interface form
solver-www-cgi.pl	completes Web interface CGI script

registered. Solver pages are also created and updated when solvers are added or modified by the ADMIN:ADDSOLVER, which calls on *make-solver.pl* with the type and identifier of the desired solver as argument. In this manner, the Web site stays current without need for the intervention of the Server administrator.

8 The NEOS Server Administrator FAQ

Included in the NEOS Server 4.0 distribution under server/lib/html/neos_faq.html is a brief list of debugging-style questions and answers for solver administrators. Many of the points in that FAQ are covered here, but we include them in the FAQ for handy reference. Also, the FAQ contains much more detailed help with responding to the Server configuration prompts when you first `make` your NEOS Server. We would like to hear your own suggestions and questions so that we can add them to our NEOS Server FAQ at `http://www-neos.mcs.anl.gov/neos/neos_faq.html`. Because we may not maintain our NEOS Server at the same site forever, you can maintain your FAQ for future administrators of your own NEOS Server.

Acknowledgments

Work on the NEOS Server first began in 1994 and expanded through the collaborative efforts of Joe Czyzyk, Bill Gropp, Mike Mesnier, Jorge Moré, Steve Wright, and others. I especially point out that Mike Mesnier, as the first NEOS Server administrator, left me much of the information included in this guide when I took over administration and development of the NEOS Server for Optimization in 1999.