Owen: A Labour Exchange for Computational Grids

Roger Peppé
Vita Nuova

ABSTRACT

To harness the power of hundreds of lightly used computers to perform large computations is an attractive prospect. Owen™ (named after Robert Owen (1771-1858)) is a component that enables reliable distribution and execution of work in this kind of environment. It is platform independent and highly modular, enabling its use in many possible scenarios, much of which is made possible by its implementation: on the Inferno OS, using the Styx protocol.

1. Introduction

Here's a one common ''compute grid'' scenario: you have a large computational job, too large for one computer to solve by itself in reasonable time, so you split it into separate tasks; give each task to be processed by a different machine; then retrieve the results, and splice them together.

Whether this will speed things up or not depends entirely on the problem. Is it possible to split the problem into multiple independent pieces? All the parameter and results data has to be shipped over the network. How long does this take relative to the actual computation time? The maximum speed-up is attainable is limited by the length of the computation time for each piece and the amount of data that needs to be transferred.

Over large networks of machines, other issues start to become important too. As a grid becomes larger, the possibility of component failure becomes a near certainty. Client machines are switched off without warning, network cables are pulled out, servers crash (or need rebooting), etc, etc.

The person with the job-to-be-solved does not want to have to care about these issues. Insofar as it's possible, they should be able to hand out the work and retrieve the results without being aware of any of the intervening chaos. This is the problem Owen is designed to solve. Human administrators hand out work, workers work, and Owen deals with the complexity in the middle.

So let us assume that you have a candidate problem that might be suitable for grid computation. This document explains how you might go about implementing it using Owen.

2. Structure

In addition to Owen itself, there are two main components involved in the computation, the *workers* that actually perform the computations, and the *task-generator* that generates tasks to be handed out to the workers.

2.1. Workers

A *worker* connects to Owen and then sits in an endless loop, asking for a task to perform, reading requisite parameter data, computing the result, and sending it back to Owen. This can be as simple as the following shell script:

```
fn runtask {
        compute_task > result
}
fn submit {
        cat result
}
```

Compute_task reads parameter data from its standard input, and writes the results to its standard output. The computation is split into two sections, one of which actually performs the task, and commits the result to non-volatile storage, the other retrieves the result and sends it to Owen. The reason for this is to avoid losing work in the event of a network outage. If a worker tries to submit its results and Owen is unavailable for whatever reason, then it will not lose the results - they will be submitted whenever it can make a connection again, even if the machine is rebooted.

The worker can have several such scripts, one for each *task type*. The worker lets Owen know which task types it has available, so it will not be handed work that it does not know how to process.

2.2. Task-generator

When a worker comes to Owen asking for some work, Owen asks the *task-generator* of the current job whether it has some work available to give it. Owen gives the task-generator some information to aid in its decision: a set of attribute-value pairs with which the worker describes aspects of itself, such as its CPU type and available software; and information that Owen has about the worker, such as its IP address and authenticated name. If the task-generator accepts the services of the worker, the two are placed in contact via a virtual link - the worker can read and write data, and the task-generator sees each read and write operation, and can respond as it sees fit. In this way, they can exchange data, as shown above, until the task is complete, and the task-generator has received the results.

3. Resilience

Owen is designed to function correctly in any of several possible failure modes. The server itself regularly dumps its runtime state, enabling it to be restarted in the event of a crash. If this happens, or if the network fails, workers will automatically try and reconnect. When they succeed, if they have the results of a computation, they will try and submit it.

One possible problem is that a particular worker is allocated a task, and then never completes it. This can be because of an problem with the worker, or perhaps because it is very slow. When Owen has sent out all the tasks for a job, Owen continues to send old pending tasks until it has received valid results for all tasks. When this happens, the job is marked complete, and any workers running duplicate tasks are asked to terminate them.

Another common failure mode is that a worker asks for a task, submits an erroneous result, immediately asks for another task, and continues in this way. If we are not careful, this can absorb large quantities of network bandwidth and server CPU time. Owen keeps track of how many tasks a worker has recently failed; if it fails more than a certain number of tasks in a row, then the worker is *blacklisted*, and will not be sent any further tasks until it is unblacklisted. This can be done manually; alternatively Owen will sporadically unblacklist old nodes, in the hope that something has been fixed in the meantime.

A particular task might have particular characteristics that mean that it always generates erroneous results. When asked to restart a task that has previously failed, a task-generator is told how many times that task has previously failed, and has the option of marking the task as permanently failed, in which case Owen will never try to start it again.

4. Security

Security in a distributed computing environment is predicated on *trust* of various levels in different parts of the system. ‘‘Will this software screw up my work?’’, asks the owner of a desktop machine that is proposed to run the Owen worker software. ‘‘How can I avoid giving out confidential task data to outsiders?’’, asks someone thinking about running a compute job using Owen. There are many properties that may be desirable, depending on the context in which the system is running. Owen tries to provide a simple

but flexible security mechanism that should be sufficient for the more paranoid, while remaining unobtrusive where strong security is not a major requirement.

In a corporate network behind a secure firewall, it is quite likely that strong security is more of a liability than an asset. It is just one more thing that might go wrong, and fixing it is often difficult, as the very security of the system can make it hard to analyse when it is not working correctly. It is possible to run Owen with authentication turned off: when running in this mode, any user on the network is potentially able to connect to Owen, start new jobs, delete old ones, and generally (ab)use the system. Given suitable corporate policies, this is not necessarily a problem. Moreover, even running in this mode, Owen is not entirely insecure from the workers' point of view. This is because the work that workers take on is entirely at their own discretion. It is quite possible for a worker to accept only a fixed set of tasks, each of which is known to act only within certain well known behavioural bounds. This can be appropriate in environments where the grid is used only for a limited number of job types, or where the owner of a client machine wants to be sure that they are not opening their machine up to a potentially malicious Owen administrator.

In an environment where all parties are *not* implicitly trusted, the first level of trust must come from the underlying system software itself. If a malicious third party corrupts the underlying software, then no amount of additional security can help matters. Assuming a system that is initially clean of such interference, the Owen security model provides assurance that it will remain that way.

5. Example

Here is an example of how one might go about using Owen to distribute a particular job. In this example, we will run the CHARMM (Chemistry of Harvard Molecular Mechanics) bimolecular simulation software across a Owen network. This software runs from the command line as follows:

`charmm '`*attr1=value1*`' '`*attr2=value2*`' ... < `*inputfile*` > `*outputfile*

CHARMM's standard input specifies the task in a custom language, the attributes set internal parameters, and it writes the results to its standard output. The input and output do not amount to more than a few kilobytes; the run time of a task can be anything from a few minutes to several hours - a perfect candidate for distributed computation, in other words. CHARMM also uses various auxilliary files, which tend to remain identical from job to job.

To start with, we have several choices as to how we might run the software, depending mostly on the desired security of the Owen network. In a network where security is not an issue, and clients are prepared to run anything that the server throws at them, we can distribute the executable directly; in another environment, it might be necessary to ask clients to install the executable for themselves (allowing them to make appropriate checks to see if they trust the thing they're going to run). In this example, we will assume the former case: we already have a vanilla Owen network, and we wish to distribute everything necessary for the computation.

The first task is to distribute the executable and supporting files to all workers. To do this, we put all the files in a directory within the Inferno hierarchy, for instance, in `/grid/inferno/tmp`, and bundle them up for distribution. (Inferno runs as a host application under various platforms; its root directory is a sub-directory of the host operating system's). In the directory containing the files, at an Inferno shell prompt:

```
% lc
charmm bench.inp rtf  parm psf coor
% fs bundle . | gzip > /tmp/charmm.bun.gz
%
```

`Fs` is a Inferno command that allows filtering and manipulation of file trees. The `bundle` sub-command archives a directory tree and writes it to its standard output. `Gzip` then compresses this. (see Section 1 of the Inferno Programmer's Manual for documention on both commands). The Inferno gzip compress utility can be rather slow, so if you're distributing a large amount of data, it is worth enabling JIT compilation, or using the host gzip utility to do it. For instance:

```
% fs bundle . | os gzip > /tmp/charmm.bun.gz
```

To actually distribute these files, we need to start an update job, so, assuming we have Owen mounted (see later section, ‘‘The Owen Interface’’ for details):

```
% start update charmm
3
%
```

‘‘`Charmm`’’ is the package name we have chosen to label the set of data files, and the update job has now started as job 3. We now provide the update job with the actual data to distribute:

```
% cat /tmp/charmm.bun.gz > /n/remote/admin/3/data
%
```

The workers should now have started downloading the bundle (which should soon appear on each client inside the directory `/grid/slave/data/charmm`). We will leave the job going indefinitely, as we need to make sure that any clients that are not currently online get the CHARMM package before starting to execute a CHARMM job.

Now we have distributed the prerequisite files for the job, we need to get the workers actually executing it. Each worker has a set of shell scripts, `/grid/slave/*.job`, which defines the set of task types the worker is willing to run. Here we will make a script to execute a new task type that will execute CHARMM tasks

```
% mkdir /tmp/charmmtask
% cd /tmp/charmmtask
% cat > charmm.job
extdir = $emuroot/$root/data/charmm
fn runtask {
        (args data) := ${unquote "{cat}}
        echo $data | os $extdir/charmm $* $args 'AUX='$extdir > result
}
fn submit {
        cat result
}
^D
%
```

We define `extdir` to be the full path to the CHARMM auxilliary files. Recall that Inferno runs within a directory within its host’s filesystem. The path to this directory is held in `$emuroot`. The worker software defines its directory in `$root` (conventionally it is `/grid/slave`).

When `runtask` is invoked, it uses the shell quoting mechanism to split its input into two sections, the per-task arguments to be passed to CHARMM, and parameter data to be given to CHARMM’s standard input. There were several possibilities here; this approach is reasonable here because neither the arguments nor the parameter data is very large. If more data, or an arbitrary set of files, was being distributed, we might use a file archive format (in some scenarios, one might wish to distribute all the auxilliary files and the executable for every task, for example).

The `os` command runs CHARMM (found in the set of files we distributed earlier). We give it the job-global arguments (the task-generator arguments given when first starting the job), the task-specific arguments (as decoded above), and we let it know where the auxilliary files live (internal file references inside the CHARMM script reference the @AUX variable, as set on its command line). The results of the computation are written to the `result` file, where they can later be picked up by `submit`.

Now we have written the script, we need to distribute it to all workers in a similar way to the auxilliary files:

```
% fs bundle . | gzip > /tmp/charmmtask.bun.gz
% start update charmmtask install /grid/slave
4
% cat /tmp/charmmtask.bun.gz > /n/remote/admin/4/data
%
```

Note that the arguments to the update job are different this time, as we need the new job file to be put into its correct directory, so the worker will recognise it.

Now to actually run a CHARMM job. The `filesplit` task-generator generates a task for each record in a file. The record format is simple: each record starts with a header line (newline terminated) of text, for instance:

```
data 300
```

The number gives the size of the data in the record (in this case 300 bytes) that follows the header.

For some jobs, one might wish to generate this with a program, such as `perl`. In this case, we'll use Inferno to do the work. In a clean directory, create two files for each task to be executed, one containing the CHARMM command-line arguments for the task, and the other containing its input script. I assume this has already been done, and the files are named *n*`.arg` and *n*`.data` respectively, where *n* is the number of the task. The following script puts them all into records that can be picked up appropriately by the script we have written earlier. This particular job has only three tasks:

```
% lc
1.arg 1.data 2.arg 2.data 3.arg 3.data
% for(i in 1 2 3){echo ${quote "{cat $i.arg} "{cat $i.data}} > $i.rec}
% file2rec *.rec > jobdata
```

The `for` loop quotes up the arg and data files for each task, and puts them in *n*`.rec`. `File2rec` writes a record for each of its argument files containing the data in that file. `Jobdata` now contains all the parameter data for the job.

Now all we need to do is make `jobdata` available on a filesystem available to Owen and start the job.

```
% cp jobdata /grid/master/myjobdata
% start filesplit /grid/master/myjobdata charmm
%
```

I've assumed here that the `/grid/master` directory is shared with the Owen server; if not, it might be necessary to copy it across in some other way (for example, by mounting a filesystem exported by the server, or using ftp or NFS).

The job should now have been started. The Owen monitor can be used to find out the progress of the job. The results will be written to a file of the same name as the input file, but with `.result` appended (in this case `/grid/master/myjobdata.result`). When the job has completed, all the resulting data will have been written to this file in the same record format as the input. We can extract the records to see the results:

```
% mkdir /tmp/results
% cd /tmp/results
% rec2file /grid/master/myjobdata.result
% lc
1 2 3
%
```

Each file holds the output data for one of the tasks in the job. In fact CHARMM itself produces several extra output files, which for simplicity we have not referred to. It would be straightforward to bundle these up with `bundle` or `tar` to see the full results of the CHARMM run.

6. The Owen Interface

Access to Owen, by both administrators and by workers is via a simple file-access protocol named Styx. All the resources that Owen makes available in this way can be accessed within the Inferno environment by opening, reading, and writing files. It is possible to translate this into other forms of access, but this is its ''native-form'', and hence I will describe its facilities in these terms.

Initial contact with Owen is conventionally made by connecting to a TCP port at a known network address. If Owen is being run in authenticated mode, the network link is authenticated at this point, and

encryption pushed on to the link (at the discretion of the server). The link is then *mounted* at some point (conventionally `/n/remote`) in the Inferno namespace. From this point on, each operation on a file or directory below this point results in a Styx message being sent to Owen via the network link, giving it the opportunity to respond as it deems appropriate. It does not have to respond immediately - it is quite normal to have several operations outstanding simultaneously, for instance when several processes are operating on the namespace at the same time.

The following example transcript from a shell console shows how access to Owen might be obtained on the Inferno command line:

```
% mount tcp!owen.somewhere.com!owen /n/remote
% cd /n/remote
% ls -l
d-r-xr-x--- M 8  admin  admin 0 Apr 13 19:58 admin
--rw-rw-rw- M 8 worker worker 0 Apr 13 19:58 attrs
--rw-rw-rw- M 8 worker worker 0 Apr 13 19:58 nodeattrs
--rw-rw-rw- M 8  admin  admin 0 Apr 13 19:58 nodename
--rw-rw-rw- M 8 worker worker 0 Apr 13 19:58 reconnect
--r--r--r-- M 8 worker worker 0 Apr 13 19:58 stoptask
--rw-rw-rw- M 8 worker worker 0 Apr 13 19:58 task
%
```

In this example, ‘‘`% `’’ is a prompt printed by the Inferno shell; text after this has been typed interactively by the user. The `mount` command dials tcp port `owen` (we assume an entry in a local network database associating this name with a numeric port number; otherwise a numeric port may be used) on IP address `owen.somewhere.com`, authenticates with the Owen server, and places the resulting namespace at `/n/remote`. The `ls` command shows the files that Owen makes available in its top-level directory.

The files owned by `worker` are available for workers to access data and make information available to Owen. Of these, arguably the most important is the `task` file. When a worker tries to open this file, Owen will not respond until it has some work available for it. When the open succeeds, the worker reads first the *server id* of the task (a unique identifier that may be used to reconnect back to Owen in the event of a connection loss, by writing it to the `reconnect` file), and then the *task type* (the name of the task that Owen would like the worker to perform). The worker may then write an identifier of its own choice that Owen will send on the `stoptask` file when it wishes to stop the task. Reads and writes are then routed through to the task-generator, as described in the previous section. Other files in the top-level directory allow the setting of worker attributes and values (`attrs` and `nodeattrs`), setting the node’s network name (useful in environments where DNS does not function properly).

All files that access the control and monitoring of Owen jobs are held within the `admin` directory. The users allowed to access this directory depend on the authentication mode in which Owen is being run. To continue the previous transcript:

```
% cd admin
% ls -l
d-r-xr-x--- M 4   rog admin 0 Apr 14 16:31 3
d-r-xr-x--- M 4   rog admin 0 Apr 14 16:31 4
d-r-xr-x--- M 4   rog admin 0 Apr 14 16:31 7
--rw-rw---- M 4 admin admin 0 Apr 14 16:31 clone
---w--w---- M 4 admin admin 0 Apr 14 16:31 ctl
--r--r----- M 4 admin admin 0 Apr 14 16:31 formats
--r--r----- M 4 admin admin 0 Apr 14 16:31 group
--r--r----- M 4 admin admin 0 Apr 14 16:31 jobs
--r--r----- M 4 admin admin 0 Apr 14 16:31 nodes
% cat jobs
93db8893 7 running 2 -1 filesplit -l /tmp/x2 test md5
669c4ca8 3 stopped 0 -1 filesplit -ln /appl/cmd/owen/scheduler.b md5sum
8f699c8c 4 stopped 1520 -1 filesplit -ln /appl/cmd/owen/scheduler.b test md5sum
%
```

Inside the `admin` directory, each current job has a numbered directory (in this case there are three such

jobs, all created by the user `rog`). The `clone` file is used to create a new job, as described below. Other files at this level describe global aspects of Owen: `group` gives information on the machines that Owen jobs will run on, `jobs`, information on all the currently running jobs, `nodes`, information on machines that have connected to Owen, while `formats` gives information on the layout of information in the above data files (for instance, the ordering of the fields on each line of the `jobs` file shown above). The `ctl` file is used to control Owen, for instance: to remove a troublesome machine from the list of machines that Owen will consider for execution of a task.

A new job is created by opening the `clone` file, which actually opens the `ctl` file inside a newly allocated job directory. Textual messages written to this control aspects of the new job; most importantly, the `load` message asks Owen to load a new task-generator for the job. The write fails if the task-generator cannot initialise properly, for instance, when it is not given appropriate arguments). When the job is no longer needed (nothing's using it, it has not been associated with a task-generator, or has been deleted) it will disappear.

```
% echo load filesplit -l /tmp/x2 test md5 > clone
% ls -l
d-r-xr-x--- M 4   rog admin 0 Apr 14 16:31 3
d-r-xr-x--- M 4   rog admin 0 Apr 14 16:31 4
d-r-xr-x--- M 4   rog admin 0 Apr 14 16:31 7
d-r-xr-x--- M 4   rog admin 0 Apr 14 17:08 8
--rw-rw---- M 4 admin admin 0 Apr 14 16:31 clone
---w--w---- M 4 admin admin 0 Apr 14 16:31 ctl
--r--r----- M 4 admin admin 0 Apr 14 16:31 formats
--r--r----- M 4 admin admin 0 Apr 14 16:31 group
--r--r----- M 4 admin admin 0 Apr 14 16:31 jobs
--r--r----- M 4 admin admin 0 Apr 14 16:31 nodes
%
```

See that a new directory (`8`) has now appeared, which represents the new job that has beenh created. Since many people may be accessing Owen at once, we usually use a little shell script to load a new job which also reports the number of the new job, to take away the guesswork (see appendix 1 for details).

```
% cd 8
% ls -l
--rw-rw---- M 4 rog admin 0 Apr 14 17:08 ctl
--rw-rw---- M 4 rog admin 0 Apr 14 17:08 data
--rw-rw---- M 4 rog admin 0 Apr 14 17:08 description
--r--r----- M 4 rog admin 0 Apr 14 17:08 duration
--r--r----- M 4 rog admin 0 Apr 14 17:08 group
--r--r----- M 4 rog admin 0 Apr 14 17:08 id
--r--r----- M 4 rog admin 0 Apr 14 17:08 monitor
%
```

The files inside a job directory enable controlling and monitoring aspects of the job, similarly to the files in the `admin` directory. Messages written to the `ctl` file can start and stop the job, alter its priority in the job queue, and change the machines that the job will be allowed to run on.

7. Appendix 1: Shell functions to control Owen

```
load std

# load a job. result is the new job id.
subfn job {
        {
                id := "{read}
                result=$id
                or {echo ${quote load $*} >[1=0]} {
                        raise 'load failed'
                }
        } $* <> /n/remote/admin/clone
}

# load a job. print the new job id.
fn job {
        echo ${job $*}
}

# send a control message to a job.
fn ctl {
        if {~ $#* 0 1} {
                echo usage: ctl job-id arg... >[1=2]
                raise usage
        }
        (id args) := $*
        echo ${quote $args} > /n/remote/admin/$id/ctl
}

# load a job, then start it.
fn start {
        id := ${job $*}
        ctl $id start
        echo $id
}
```

8. Appendix 2: A short primer on the Inferno shell

The Inferno shell has been used in this document to provide various pieces of ‘‘glue’’ script. This section gives a quick tour through some of its features. For a more complete reference, see the *sh* manual page, in Section 1 of the Inferno programmer’s manual.

8.1. Commands

A shell command is a simple sequence *words*, separated with space or tab characters. The first word in the sequence names the command to be executed; the others are passed as arguments to that command. If a word contains no special characters, it represents itself, e.g.

```
echo hello world
```

which executes the command ‘‘`echo`’’, with two arguments: ‘‘`hello`’’ and ‘‘`world`’’. The following characters are special:

```
# ; & | ^ $ ‘ ’ { } ( ) < > " = * ? [
```

8.2. Quoting

Quoting prevents the shell from interpreting special characters. Quote a word by surrounding it with single quote marks. Single quotes inside the word are doubled, so `'I''m going away'` represents the literal text `I'm going away`.

Wildcards try to match a pattern against filenames in the namespace. ‘‘`*`’’ matches any sequence of

characters in a filename; ‘‘`?`’’ matches a single character, and `[`*chars*`]` matches a single character containing any of the letters mentioned in *chars*. If any matching filenames are found, then they are used instead of the original word, otherwise the word is left unchanged.

8.3. Input/output

A shell command has access to three streams of data, commonly referred to as *standard input*, *standard output*, and *standard error*. In shell scripts, these are referred to by their file desciptor number: 0, 1 and 2 respectively.

To make a command’s standard input come from a file, use:

command `<` *file*

To send a command’s standard output to a file, use:

command `>` *file*

This will create *file* if necessary, and truncate it otherwise. To send a command’s standard error to a file, use:

command `>[2]` *file*

Despite their names, any of the above streams may be bi-directional.

command `<>` *file*

opens the standard input both for reading and writing. Multiple redirections are allowed on a single command; the shell will apply each one in sequence.

Multiple commands can be connected together with a *pipe*:

command1 `|` *command2*

This connects the standard output of *command1* to the standard input of ;*command*2 anything that is written to the standard output of *command1* can be read by .*command*2 When *command1* exits, *command*2 will see end-of-file.

8.4. Command Blocks

A number of commands may be stuck together inside a braced *command block*, in which case they are each executed in sequence. For instance:

```
{
        sleep 10
        echo hello world
}
```

would sleep for 10 seconds and then print the string ‘‘`hello world`’’ on its standard output. A semicolon may also be used instead of a newline as a command separator, so the following behaves identically:

```
{sleep 10; echo hello world}
```

An ampersand ‘‘`&`’’ may also be used, which causes the preceding command to be run in the background; execution of subsequent commands in the block continues without waiting for it to finish.

A braced block may also be used as an argument to a command - in this context, it is treated as a normal word. So for instance:

```
echo {sleep 10; echo hello world}
```

is perfectly valid, and just prints the command block, without executing any of the commands inside it.

8.5. Variables, lists and concatenation

A shell variable contains a sequence (list) of words:

```
x=how now 'brown cow'
```

The variable `x` nows holds the three words `how`, `now`, and ‘‘`brown cow`’’, and can be referred to as `$x`. For example:

```
echo $x
```

would print ‘‘`how now brown cow`’’.

The ‘‘`:=`’’ operator is similar, except that the variable is only accessible within the current command block, and inside commands that it invokes. For instance, after:

```
{
        x := a b c
}
```

the value of `x` will be unchanged.

A list of words can also be denoted with brackets. So:

```
x=(how now 'brown cow')
```

is exactly equivalent to the above assignment. It is perfectly usual to have a list containing no words at all. This is the default value of all shell variables.

A list containing a single element may be *concatenated* with another list with `^`. For instance:

```
echo ho^$x
```

would print ‘‘`hohow honow hobrown cow`’’.

Wildcards are expanded *after* list concatenation, so for instance:

```
echo (/bin /dev /usr)^/z*
```

will echo the names of all files beginning with ‘‘`z`’’ in any of `/bin`, `/dev` or `/usr`.

The standard output of a command can be converted into a list with the backquote and doublequote operators. For instance:

```
x="{echo hello world}
y='{echo hello world}
```

The first form produces a single word containing the entire output. The second form splits up the output into separate words, using spaces, tabs and newlines as delimiters. In this example, `$x` will end up containing one word, and `$y` with two.

You can count the number of elements in a variable *x* with `$#`*x*. For example:

```
v=a b 'c d'
echo $#v
```

will print ‘‘`3`’’.

8.6. Loadable modules

The above sections have described almost all of the fundamental features of the Inferno shell. Other functionality is provided through the use of extension modules that can be loaded into the shell on demand. The most important of these is `std`, which defines some commands that give the shell programmability. To load a module:

```
load std
```

To find out which commands have been made available by the currently loaded modules, use the `loaded` command.

When a shell command terminates, it yields an *exit status*, depending on its success or failure. This

is held in the shell variable `$status`. This is empty if the command succeeds, and holds some indication of the problem if the command failed.

Several of the commands defined by `std` use this to provide control-flow. For instance:

```
if {diff x1 x2} {
        echo files x1 and x2 are the same
}

for i in 1 2 3 4 {
        echo i is now $i
}

while {! mount tcp!myserver.home.com!1234 /n/remote} {
        sleep 20
}
```

Note that unlike most shells, these commands have no special syntax associated with them - the command and its arguments are just a sequence of ‘‘words’’ (recall that a command block counts as a word). This means that, for instance:

```
if {diff x1 x2}
{
        echo files x1 and x2 are the same
}
```

will not have the desired effect!

There are two types of commands that can be defined by loaded modules: normal commands and commands that result in a list. Normal commands are invoked in the conventional way. Commands that result in a list are invoked as `${`*command*, for example:

```
echo ${split ':'  'Joe Bloggs:128:red'}
```

This invokes the named command, which returns a list of words that are substituted in place of the `${}` block. In this case, the `split` command splits up its second argument at any character in its first argument, so this command would print ‘‘`Joe Bloggs 128 red`’’, (echo will be passed three arguments).

8.7. Quote and unquote

The `quote` and `unquote` list-returning commands are used to bundle up a list into a single word, and to reverse this process, respectively. This enables shell commands and values to be passed across the network unscathed. The quoting convention used by the Inferno shell is also understood by most Inferno device drivers and many Styx services (for instance control messages to Owen are written in this way). For instance:

```
echo ${quote how now 'brown cow'}
```

produces the string ‘‘`how now 'brown cow'`’’, and

```
x=${unquote 'how now ''brown cow'''}
echo $#x $x
```

produces ‘‘`3 how now brown cow`’’. Any sequence of words will be quoted by `quote` in such a way that exactly the same sequence of words will be produced when the resulting word is given to `unquote`.

8.8. Further reference

For more complete information on the Inferno shell, see the various manual pages in Section 1 of the Inferno Programmer’s manual `sh,( sh-std`, `sh-expr`, `sh-arg`, `sh-string`, `sh-file2chan`, `sh-tk`, et al). There is also a document describing some of its features: ‘‘The Inferno Shell’’. All of the Inferno documentation can be found on-line at: `www.vitanuova.com/inferno/docs.html`.