Distributed Command Line Interface

Anonymous

December 6, 2001

Abstract

This paper presents a design for a command line interface that allows any user to seamlessly distribute processes among multiple machines with very little underlying architecture. Our distributed shell, *distsh*, is designed to be a suitable replacement for any common Unix shell. It provides all the expected functionality of standard command line interfaces but with an added layer of interpretation that allows the user to execute and control remote processes. *distsh* is designed to be lightweight, secure, portable and usable by unprivileged users.

Distsh works by taking advantage of expected shell behaviors such as job control and input/output redirection, kernel support for pseudo terminals and by exchanging network pipes for kernel pipes. The shell parses specially formed commands and uses an existing transport mechanism to connect to a remote machine to execute the command. Remote processes are presented to the user as jobs, with the shell transparently passing signal and control messages between the two machines. Redirection is performed by tunneling the data from processes input and output over the transport. Our design is straightforward, easy to use, and requires no kernel modifications or administrator support.

1 Introduction

With the decreasing cost of computers, we are moving from a one-computer-many-users scenario to a many-computers-one-user one. Users are likely to have network access to multiple machines, and will want to utilize those resources as effectively as possible.

Distributed storage resources have been well-researched, resulting in the development of systems such as NFS, AFS and SFS. These systems work by abstracting the concept of a file in the OS kernel, making remote and local files appear identical to users. Systems that similarly abstract process threads exist, for example Sprite[5], but often require serious kernel modifications and are therefore not useful to the average users. We have found that they are often over-ambitious, for example, including process migration, or too specialized, for example, only functioning on Beowulf systems[6]. In order to be widely accepted, we believe that any solution to the problem of managing distributed processes should not require modifications to the existing operating system.

In designing and implementing *distsh* we decided to focus on usability from the standpoint of the average Unix user. We determined that users with access to multiple networked computers use essentially two variants of the same method for managing processes on remote machines. In order to execute a process remotely, the user begins by logging in to the remote machine using some preexisting transport protocol (for example, *ssh*, *rsh*, *telnet*, etc.). The user then requests that the remote terminal execute the process.

One variant on this approach is when the user keeps the terminal window open and types all job control and execution commands in that window. The user has access to all the process control available in the shell, but ends up with a separate terminal open for each machine, cluttering their workspace. Also, the user cannot easily perform I/O redirection between processes on the local and remote machines.

Another variant is to use built in command execution, such as that in *ssh*. Built in command execution has the benefit of allowing users to easily perform I/O redirection on the command line. This method is very useful but constrains the user to that transport protocol, requires them to enter a password for each time, has no simple environment control, and has somewhat unreliable job control (placing remote processes in the background is not fully functional in most version of *ssh*).

Although it is possible to use network pipes to redirect I/O in the first variant, or to add remote execution for every transport protocol in the second, we decided to solve all these problems at once by hiding them inside a shell, allowing us more flexibility and more functionality.

In this paper we will first analyze some of the existing solutions that have been developed with remote process control in mind. Afterwards, we will outline the interface and usage patterns we have designed. Then we will describe the structure of most shells and how we used that structure to create the illusion of remote process control. Finally, we discuss the actual implementation of *distsh*, as well as where more functionality could be added and the conclusions we have made about our approach and implementation.

2 Related Work

Within the clustered computing community, a few similar command line interfaces exist for managing remote processes. There is a *dsh* already used to manage Beowulf clusters. However, Beowulf machines only represent a small fraction of networked hosts, and *dsh* does not allow users to control machines outside of a single cluster. *Pdsh* (parallel distributed shell) is another system designed to better support running the same commands on multiple hosts, and is used in the management of super-computing clusters. In contrast, we aim to create a shell for low-privilege users and administrators alike and for heterogenous networks.

Network shell[4], also known as *nsh*, is a commercial product and has goals more similar to ours, but approaches them differently. Designed as a system administration tool, users must run specialized daemons that perform the function of a transport layer and interpreter. The system does not support X-forwarding or redirection and every remotely executable program must be recompiled with special libraries. Without recompilation only non-interactive text-based programs can be run remotely. However, a remote file system is more seamlessly supported, allowing users to open remote files from within an application. Since we provide only a shell, we cannot offer this service, and require the user to run some network filesystem in order to share files from within programs. We also believe *nsh* is too bulky, providing support for Windows as well as Unix computers, and requiring binary recompilation. *Distsh* is meant to be lightweight and portable and not require running any unfamiliar daemons.

The idea of multiplexing network connections is not a new one, *Fsh* is an implementation of an *ssh* multiplexer. Merging the functionality of *fsh* into *distsh* could make *distsh* easier to use, and more transparent, without giving up security. *fsh* is, however, not a shell but a front end to *ssh*. It is certainly a step in the right direction, but it does not have the cohesive design of *distsh*.

3 Usage

We found the most difficult part of designing this system to the user interface. Many possible functional additions were discarded because they would have complicated use and could have confused novice users. Instead, we decide to stick to a simple design that kept to the traditional shell interface yet is open to additions.

Because we are building a user interface, usage patterns are very important. What follows is a discussion of the changes to the shell that will be visible to the user, followed by a short example session.

3.1 User-visible Modifications

The goal of *distsh* is to have it work as much as possible like a standard shell, and provide only necessary interface changes and extensions. Furthermore, it is important that those modifications be as consistent as possible with the existing interface, so that *distsh*'s behavior will not surprise users.

The primary modification made to the shell is that in any location where the user was previously able to specify a command, the command may be prefixed with user and host information indicating a remote execution. The format for specifying a remote execution is "`[user@]host:[/path/]program`." So, for example, if a user wanted to run *emacs* on the machine `blood` as the user alice she would type "`alice@blood:emacs`" at the command prompt. Even though the process is remote and will not show up when the user types "`ps`," it will show up in the "`jobs`" list, which is completely internal to the shell. We intend users to interact with remote processes using the job abstraction, a notion familiar to anyone who has ever used a job-control enabled shell (which most shells these days are). By using jobs instead of processes in the interface, we avoid having to manipulate the process tables in the kernel. To send a signal to a remote process, users can look up the corresponding job number via the "`jobs`" command and then type "`kill %n`" where n is the job number. The same is true for putting processes in the foreground or background, with "`fg %n`", etc.

The user may perform normal I/O redirection using '`<`', '`>`', '`|`' to pipe data to or from a remote or local process to another remote or local process or to a remote or local file. Because the "`program > file`" construct is completely internal to the shell, it is trivial to allow file redirection to specify remote files. Therefore, running the command "`alice@blood:cal > alice@sweat:output`" will create the file "output" on the machine sweat.

An important distinction to make here is that the file must be one that is specified to the shell and not to some other program. So, for example, "cat < foo > blood:~/bar" will essentially copy the file foo to the file bar on the host blood, but "cp foo blood:~/bar" would copy the file foo into another file called "blood:~/bar" on the local machine. This behavior occurs because "foo" and "blood:~/bar" are arguments to the the program *cp*, passed without any modification by the shell.

We also made a few more modifications to built in shell commands that we thought would be useful to users. The *cd* command is built in to the shell and is therefore easy to overload. We allowed the argument to *cd* contain a username and host part, to refer to a directory on a remote machine. When a command or file is specified without a host/username, it is run on the host and under the username specified in the current working directory.

Handling environment variables requires special attention, since they are often for machine-specific purposes. We use a *tcsh*-style *setenv* command for environment variables (setenv foo bar sets the variable foo to the value bar). In our design, *setenv* has two flags: -a which sets the variable on all hosts, and -h which takes an argument of the form "user@host" and sets the variable on the specified host for the specified user.

3.2 Example Session

In this session, 6.824 student Alice takes care of a few tasks from her workstation using *distsh*. Alice starts out running *pine* on her local workstation to read her email:

```
distsh$ pine
```

Using control-Z, Alice suspends *pine* and proceeds to go finish her 6.824 paper, which is stored on a machine in LCS. She changes her current working directory to that machine and runs *emacs* to edit the paper.

```
distsh$ cd blood.lcs:~/6.824/final
distsh$ pwd
alice@blood.lcs.mit.edu:/home/to0/alice\
/6.824/final
distsh$ emacs paper.tex
```

Emacs will run on the machine blood, and will be editing the file paper.tex which is stored on blood. Alice suspends *emacs* and begins to typeset her paper. Discovering that the *fig2dev* on blood doesn't support EPS, Alice runs the *fig2dev* she knows works on one of her personal machines before running *latex*. job.fig and job.eps are both files in the current working directory on the current machine (blood), but *fig2dev* will run on the machine dorm.

```
distsh$ dorm:fig2dev -L eps < job.fig > job.eps
distsh$ latex paper.tex
```

Wanting a hard copy of her paper to look at, and being near an Athena printer, Alice sends the file to athena.dialup and prints it from there.

```
distsh$ dvips paper.dvi -o - | athena.dialup:lpr
```

However this command does not succeed thing because the printer is not specified correctly. This problem could of course be fixed with a command line option to *lpr*, but Alice decides to set an environment variable, which will last the length of her session.

```
distsh$ setenv -h athena.dialup PRINTER helios
```

The command sets the PRINTER variable on only the host athena.dialup. Finally, Alice realizes she has forgotten the turnin procedure for final papers, and continues her suspended *pine* in order to find the relevant message.

```
distsh$ jobs
[1]    Stopped       pine
[2]    Stopped       emacs paper.tex \
                     (alice@blood.lcs)
distsh$ fg %1
```

4 Design

First we describe the structure of a traditional shell, paying special attention to the challenges in managing remote processes. Then we discuss the design of *distsh* in parts, considering the solutions to these problems.

4.1 Architecture of a Traditional Shell

Most all Unix shells, including ours, share the same underlying structure. Run interactively, the shell takes control of the terminal, ignores signals, and presents the user with a prompt. Lines of text are read from the terminal and parsed by the shell into commands, arguments and built-in shell constructs. Jobs are defined as processes that are tied together via pipes, i.e. '|,' '>,' '<,' etc. Every command is given its own process ID, which is known to the kernel, but jobs are a built in shell construct, as are I/O redirection pipes. The kernel does, however, provide job and terminal control to a shell via groups and sessions. Every process has a group ID, usually equivalent to its process ID. When that process is in a job, the group ID is the same among all processes and usually equal to that of the "head" process. Job numbers are just a mapping to group IDs that is performed inside the shell.

I/O redirection in the shell is performed using kernel pipes (created with the pipe(2) function call). If the input to a command is redirected from the output of another, the shell creates a pipe, and passes the producer half to the first command and the consumer half to the second. More specifically, after the shell fork(2)'s, but before it exec(3)'s, the shell dup2(2)'s the pipe into the standard out and standard input (respectively) file descriptors. If the job is in the foreground, it is given the terminal via tcsetpgrp(3), otherwise, the terminal remains in control of the shell. When the foreground job is suspended or killed, the terminal returns to the shell. Shells also provide various built-in commands for controlling jobs such as *fg*, *bg*, and *kill*.

4.2 Challenges in Distributing Processes

In creating a distributed shell, the main shell process, which is responsible for interacting with the user, must maintain the same level of awareness about the remote processes it starts as it would if they were its children. However, these processes might be on other physical machines or running under the privileges of other users, and so cannot themselves be children of the shell.

Processes run from a shell by the user will have a controlling terminal. This terminal may be the terminal that they are outputting to and reading from, or it may be a terminal with which they have no other association (such as for the middle jobs in a pipe chain). It becomes important in job control, as the user may ask the terminal to send signals to jobs, such as SIGTSTOP which is usually sent by pressing control-Z. In building *distsh*, we must associate remote processes with the appropriate local terminal, just as if they were local processes.

Processes may be aggregated for easier management, into units known as process groups. The most common method is the pipe chain, where the output of one command is fed as the input to another. Pipe chains, and process groups in general, can have any number of elements. Signals from the controlling terminal will be delivered to the group as a whole, so that all commands will be stopped and resumed together. *Distsh* must maintain a comparable association between remote processes scattered across multiple machines but in the same job.

4.3 Commheader

When a process is requested on a remote machine, either alone or as part of a job, the main *distsh* process spawns a commheader process. The commheader a stub is responsible for managing communication between the remote process, the main *distsh* process and the local process group. Remote communication can occur over an arbitrary transport, though only *ssh* will be supported at first.

The commheader has a connection open back to the *distsh* process over which it can inform *distsh* of the status of jobs. Because remote processes are not children of distsh, the standard method for tracking their status (SIGCHILD and waitpid(2)) is not sufficient. In addition, whenever distsh would wait on its children it will also check this connection for message. Only one form of message is supported at present, defined in Table 1.

Figure 1 shows the position of the commheader in the shell while running a job spread across local and remote processes. When a remote process is requested by the user, the shell forks and runs the commheader. The commheader starts the local place holder process (Section 4.4) and tells distsh that the pid so that the shell can keep track of what it thinks are its processes.

4.4 Placeholder Processes

Because a process can only be in a single process group, and because the shell likes to deal with local processes, the commheader not only runs the desired process on a remote host, it also creates a very simple process to be the placeholder. This process is responsible for receiving data from the user at the terminal or from the other processes in a pipe chain, for sending data along to the next process in the pipe chain, and for receiving signals from the user, signals to the process group.

When the placeholder receives signals, they are noticed by the commheader (its parent) and in turn forwarded to the remote process where they take effect.

4.5 Remote Connections

Remote connections are treated as bidirectional pipes between the a commheader and a slave distsh. The slave distsh is most often on a remote machine,

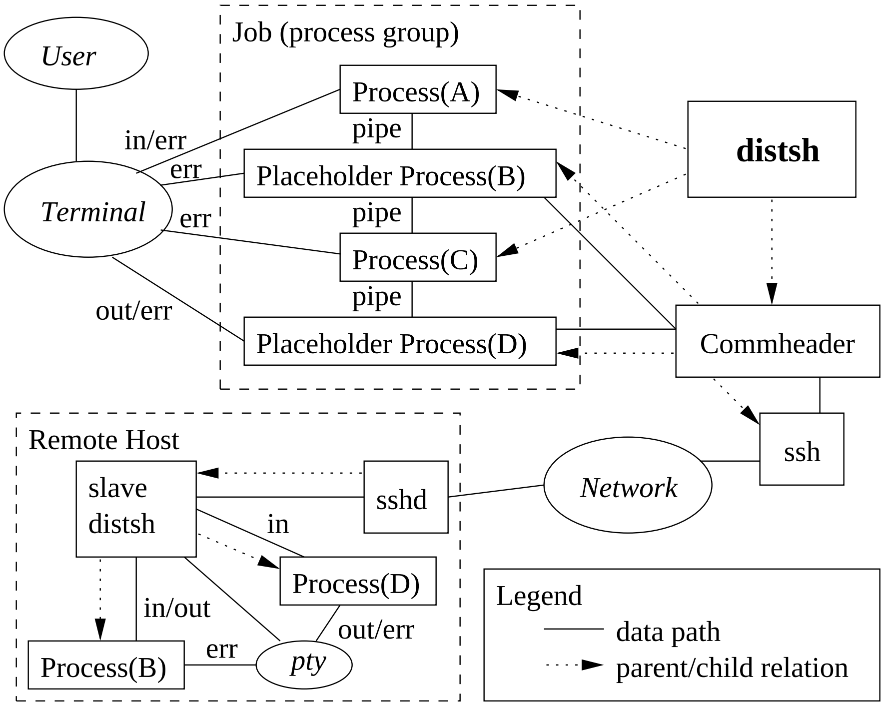

Figure 1: Job combining local and remote processes

Name	Format	Description
Status	status *pid status*	Update the status of *pid* as if waitpid(2) had returned *status*

Table 1: Messages sent from the commheader to distsh

though it could be on the local machine (most useful if it is running as a different user). In the future we would like to support multiple connection methods, but for now we choose only ssh because of its wide acceptance, standardization, and security.

Once the remote connection is established, the commheader and slave speak the a simple protocol. The protocol is human readable, to ease debugging, and is based on the protocol used by *fsh*, an existing ssh multiplexing tool[1]. The types of messages broadly fall into 2 categories: control messages and command output. Control messages allow slaves to communicate the status of processes to commheaders and commheaders to send signals to the remote process, and to start new processes. Data stream messages are simple, carrying a stream of data in one direction or the other. They are used to implement pipe chains and to get input and output from the users terminal. All the types of messages are shown in Table 2. Most messages have as a parameter a connection id (*cid*) which uniquely identifies the connection, and thus the remote process, to which that message belongs. Some messages have a stream parameter, which is a string of 8-bit characters prefixed with a length.

When required, the remote slave distsh will also provide pseudo-terminal support, such that remote processes can have a fully functional terminal driver. The master side of the pseudo-terminal will be placed in packet mode and connected over the multiplexed ssh stream. libpty[8] is used to gain a more operating system independent interface to pseudo-terminals.

4.6 Job Control

Job control for jobs which exist on a single host is handled in the traditional way. All processes in the job are placed in a single process group. The foreground job is given the terminal as its controlling tty, and only processes in that job can read the terminal.

With a remote job running on a single host, a pty is used to connect the process to the terminal. When

Name	Data	Origin	Meaning
new	*cid program*	Master	Start *program* with the new connection id *cid*.
status	*cid status*	Slave	Specifies that the process now has status *status*.
signal	*cid signum*	Master	Sends the signal *signum* to the process.
stdin	*cid stream*	Master	Data sent to stdin of the process.
stdout	*cid stream*	Slave	Data from stdout of the process.
stderr	*cid stream*	Slave	Data from stderr of the process.
setenv	*name value*	Master	Set environment variable to value.

Table 2: Currently defined message types

the remote job receives a signal such as SIGTSTOP, the slave shell reads the status of that job and notifies the master shell through the commheader.

Remote jobs spread across multiple hosts present a more complex problem. Pipes within the job must be replaced with distsh data streams, with the master shell responsible for moving the data between machines. The standard Unix system of pseudo-terminals is used to give remote jobs read and write access to the users terminal, when appropriate. However, signals that a generated by the terminal driver based on user input (such as SIGINT and SIGSTOP) must be caught by the master shell and passed as a control message to the appropriate slave shell which can then signal the members of the job that are on its host.

5 Future Work

Distsh is a prototype shell and shows that this tool can be built and can be useful. There are a number of directions in which it can be extended to increase efficiency and utility.

5.1 Hierarchical Connections

Sometimes a user can connect from host $A$to $B$and from $B$to C, but not from $A$to C. This failure may result from an intervening firewall, trust relationships or the lack of a common protocol. This situation comes up frequently in the corporate and academic world where one user may cut across multiple management domains.

Hierarchical connections would allow users to run commands of the form `B:C:command` (run command on Cpassed through B) or even to specify pathways to reach certain hosts. The system described above does not make this usage impossible, but explicit support would be nice.

5.2 Stream Routing

When a process is started on a remote machine, all of its input and output runs over streams which return to the machine running the master shell. While this waste of bandwidth is acceptable in most cases, sometimes the master shell will be running on a bandwidth starved workstation, for example on a dialup link. In this case, bandwidth back to the host is at a premium and should not be used unnecessarily.

As a first optimization, data streams between processes on the same host should be detected and optimized to not use the network at all (simple local pipes are sufficient). This addition is possible because all processes on a host are children of the local slave distsh. However, relations between processes probably need to be detected in the master distsh because the appropriate information is not available to the slave distsh.

Sometimes the user might be moving large amounts of data between two remote machines, for example two machines in the lab while the user is at home. In the current implementation there is no good path over which to send this data. Allowing the user to open additional connections between two slave hosts over which data may be sent is a further optimization. Data could be routed over these connections either automatically or only by user request, and these routes could be static or dynamic.

5.3 Alternate Transports

Some users don't have access to *ssh*, or want to use Kerberized or home-grown services. Although the system uses ssh as a transport, tt should also work with other similar channels, such as *lsh*, *fsh* and *rsh*. Extending it to work with other channels would be valuable to users.

Of particular utility might be working with the su program. While su is not a network command, it would allow users to run commands as other users on the local machine using the familiar syntax of distsh.

6 Implementation Status and Conclusion

As of this writing, the core of the system is implemented. Local and remote processes are created by the shell, and remote processes are managed over simple bidirectional pipes. We have verified that job control and redirection using the commheaders does work. However, time constraints and the complexity of internal shell structures forced us to pass on hooking in the ssh multiplexing system, and so remote processes currently only run on localhost. We are confident that these issues are temporary, as even multiplexors like *fsh* are simple and can be integrated into *distsh* using just an exec(2) system call, though they do not offer the full functionality we desire. We do not anticipate any further difficulties as we proceed with implementation, and we hope that the system, once it is extended to a usable state, will be a strong base for extension.

During our design phase, we experimented with many different UI possibilities, including interaction with NFS and overloading of more environment variables. We determined, however, that despite some of the clear advantages, much of our functionality could only be used by used by power users and could severely hinder novice users. Its is still not clear how novice users will handle broken transport connections, for example. We also decided that if we were to build *distsh* with lightweight design in mind, then trying to interact with external projects like NFS could easily turn out for the worse. We strongly suggest that others attempting to build similar systems should spend a majority of their time researching the interface. Clearly, for projects like *distsh* to be successful outside academia, the user interface of common shells must be examined more carefully.

In conclusion, distsh makes the full power of networked Unix systems more available to average users. By integrating remote process invocation and management with the shell, and providing a consistent interface to local and remote computation, we hope that it will increase utilization of these resources. We also expect the system to be useful to power users and system administrators who currently make full use of the networked computers at their disposal but who would prefer a simpler, cleaner interface.

References

[1] Per Cederqvist. fsh — fast remote command execution. World Wide Web, Dec 2000. http://www.lysator.liu.se/fsh/.

[2] Free Software Foundation, 59 Temple Place — Suite 330, Boston, MA 02111-1307, USA. *The GNU C Library Reference Manual*, 0.10 edition, Jul 2001. http://www.gnu.org/manual/glibc-2.2.3.

[3] Jim Garlick. Pdsh — parallel distributed shell. World Wide Web, Jul 2001. http://www.ecst.csuchico.edu/ garlick/pdsh/.

[4] Blade Logic. Network shell (nsh). World Wide Web, 2001. http://www.bladelogic.com/-products_services/network_shell.html.

[5] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch. The Sprite network operating system. *IEEE Computer*, 21(2):23–36, February 1988.

[6] Jason Rappleye. dsh — distributed shell. World Wide Web, Mar 2000. http://www.ccr.buffalo.edu/dsh.htm.

[7] W. Richard Stevens. *Advanced Programming in the UNIX Environment*. Professional Computing Series. Addison-Wesley Publishing Company, June 1992. ISBN 0-201-56317-7.

[8] MIT Kerberos Team. Kerberos 5 release 1.2. World Wide Web, Feb 2001. http://web.mit.edu/kerberos/www/krb5-1.2.