# BEOSH: THE BEOWULF CLUSTER SHELL

by

Mason E. Vail

A project

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Computer Science

Boise State University

August 2006

The project presented by *Mason E. Vail* entitled *BEOSH: The Beowulf Cluster Shell* is hereby approved.

| | |
|---|---|
| Amit Jain, Advisor | Date |
| John Griffin, Committee Member | Date |
| Tim Andersen, Committee Member | Date |
| John R. Pelton, Graduate Dean | Date |

dedicated to Michael W. Vail, my inspiration and hero for as long as I can remember

# ACKNOWLEDGEMENTS

I could not have finished this project without the support of my family. I cannot thank my wife, Brenda, or my kids enough for their unwavering love as I have neglected them horribly over the last couple years. Thanks also to my parents, not only for providing me with an exceptional example to live up to, but for always being my biggest cheerleaders.

I must thank my advisor, Dr. Amit Jain, for introducing me to the wonderful world of parallel and distributed computing. I may be cursed now with a lifetime of dissatisfaction with mere single-user workstations, but at least I have been given excellent instruction in playing with the really big toys when they are available. It has been a pleasure working with him. Thanks also to Luke Hindman for an invaluable fifteen minutes of consultation about client authentication.

Thank you to everyone at Crowley Davis Research, Inc. for the incredible support I have received as I have split my time between work and school. It has meant more to me than I can express.

Perhaps ironically, I must thank Micron Technology, Inc. for giving me the boot, kickstarting me to fulfill my dream of grad school. Thanks also to Uncle Sam for providing the Micron National Emergency Grant that funded most of my coursework and to Idaho Job Services for helping me get that money. We killed uncounted trees in triplicate paperwork, but I couldn't have done it without them.

This material is based upon work supported by the National Science Foundation under Grant No. 0321233. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

# AUTOBIOGRAPHICAL SKETCH

Mason Vail has had ten years of post-secondary education as of this writing and has remarkably little to show for it, due in large part to an exceptional inability to limit himself to courses that combine to fulfill the requirements of any particular degree. In spite of himself, however, he has managed to pick up a B.A. in Biology at Northwest Nazarene College, Nampa, Idaho, now Northwest Nazarene University, and a B.A. in Computer Science, also at NNU. Despite (or thanks to) his nearly debilitating lack of focus, he has spent a great deal of time trying to find a way to make a career of being a student. Unfortunately, it seems professional students make somewhat less than necessary to sustain a family.

While pursuing the second B.A., Mason enjoyed three years of part-time teaching in the NNU Biology Department. Following graduation, he worked just short of three years as a software engineer at Micron Technology, Inc. in Boise, Idaho before The Big One of 2003 which left him with a great deal of free time in which to try his hand at slowly earning a Master's degree. Along the way he joined Crowley Davis Research, Inc., a biologically-inspired computing research and development group in Eagle, Idaho. Mason now enjoys work each day, probably because it is so like being in school and prolongs the completion of his current academic endeavors.

Through all of this, his wife, Brenda, and children, Austin and Ashtyn, have somehow continued to love and support him for which he is eternally grateful.

# ABSTRACT

The Beowulf Cluster Shell (`beosh`) has been created to provide cluster users with a Single System Image (SSI) by distributing individual commands or pipelined jobs over available nodes without requiring the user to be aware of where or how jobs are distributed. Job control of distributed jobs enables management of multiple concurrent remote jobs from a single shell session. In addition to SSI features, `beosh` supports parallel job execution through `pdsh`, a commonly used parallel-only cluster shell. In either distributed or parallel mode, `beosh` can limit nodes to those reserved through a system like the Portable Batch System (`PBS`).

This shell seeks to address the lack of a full-featured, portable cluster shell that remains despite increasing availability and use of dedicated, high-performance computer clusters. Many cluster "shells", for example, are actually scripts for local shells executing a sequence of remote commands for user convenience. At the other extreme are specialized cluster operating systems requiring commitment of the cluster to a particular operating system and its included utilities. Between these extremes, there are no choices for a general-purpose cluster shell comparable to even a basic single-system shell.

Though opportunities for improvement remain, `beosh` is already a usable shell providing a foundation for development of a full-featured, general-purpose shell for Beowulf clusters.

# TABLE OF CONTENTS

# LIST OF FIGURES

# Chapter 1

# INTRODUCTION

## 1.1 Rationale and Significance

The Beowulf Cluster Shell (`beosh`) seeks to address the current lack of a general-purpose cluster shell. Despite increasing availability and use of dedicated, high-performance computer clusters, basic software utilities including shells are not as readily available for clustered systems as for individual systems.

The command shell is, perhaps, the most familiar and often-used application by single-system users. The command shell takes user commands and executes programs on the user's behalf, shielding the user from underlying system details. Often, command shells also have convenient, built-in functionality to help users manage the processes they have requested. Numerous command shells exist for single systems, differentiated by their syntax and feature set.

In a cluster environment, however, there are fewer options. Most often, users log in to a single node of the cluster and interact with a single-system shell. When the user wishes to issue commands to other nodes, he or she must explicitly execute those remote commands using protocols like `rsh`, `ssh`, or `telnet`. While this is sufficient for

some purposes, it is severely limiting and forces the user to be intimately involved in the distribution of jobs to specific cluster nodes. In addition, assigning multiprocess, pipelined jobs to a single remote node fails to take advantage of the system's potential by distributing multiple processes across available nodes in the cluster.

A more appealing ideal is the Single System Image (SSI) where the user is not required to be aware of the nodes in the cluster. From the user's perspective he or she is simply connected to a single, powerful, multiprocessor system. Commands given to an SSI shell could execute anywhere in the cluster and, to the user, they appear to have executed locally. An SSI shell should shield the user from cluster details just as a single-system shell shields the user from underlying system details.

In addition to distribution of commands across a cluster, it is also sometimes desireable to execute the same command on multiple nodes in parallel. While there are shells in common use that support parallel execution of commands, there are none I am aware of that provide distribution of commands or job control as a user would expect from a single-system shell without requiring commitment to a specialized cluster operating system and its accompanying set of utilities.

The primary goal of the Beowulf Cluster Shell (`beosh`) project is to create a high-quality, production-ready shell for use in an unspecialized Beowulf cluster environment that provides a user with a Single System Image and the same basic job control and process pipelining capabilities a user would expect from a familiar single-system shell like Linux-GNU `bash`. When needed, the shell should also allow concurrent

execution of the same job across multiple nodes.

## 1.2 Prior Work

Several approaches are represented in existing cluster shells. The most basic approach is to use a shell script to automate the distribution of commands to nodes using the same remote protocol commands the user would otherwise have to enter repeatedly. Examples of this approach are `dsh` [7] and `dish` [9]. While they do provide the capability to enter commands once and have them executed on multiple nodes, they cannot easily offer valuable features like job control or distribution of pipelined processes.

A far more complicated approach has been to create a specialized cluster operating system and accompanying utilities. Such systems are purposely created to present the user with a Single System Image and can provide not only for the parallel execution of commands but for transparent distribution of processes and full job control. While several examples of this approach have been developed [1], such as MOSIX [6] or GLUnix [4], these solutions require commitment of all clustered computers to that specialized operating system.

A third approach is through a distributed user-level shell. Such shells do not dictate a specific operating system but can provide many of the same features as shells in specialized cluster operating systems. Unfortunately, few such shells exist. The most widely used distributed shell of this type today is `pdsh`, the Parallel Distributed Shell [3]. Although it is a more powerful and refined application than any script, `pdsh`

is still limited to parallel execution of commands like the script shells and offers no job control features other than killing the current job. Two more ambitious projects to explore user-level distributed shells were Shell over a Cluster (`SHOC`) [8] and `distsh` [2]. Although these shells added more sophisticated process distribution and job control, neither project resulted in a production-ready shell for general use.

The `SHOC` project made modifications to the Linux-GNU `bash` shell to allow it to run unmodified existing programs on a cluster and, most notably, provided dynamic cluster-wide load distribution and balancing including process migration. In addition, `SHOC` allowed semaphore control from the user's login host, distributed shared memory control, and added a forall construct for scripting with distributed nodes. Several desirable shell features missing from `SHOC` were job control, signal handling, and support for processes using pipes and sockets due to the difficulty of handling such connections in the presence of process migration. Although the professor who oversaw the project indicated willingness to share the `SHOC` code, he was unable to locate a copy when contacted.

`Distsh` was designed to allow users to explicitly execute commands on the cluster node of their choice to make use of resources that may be specific to that node, without having to explicitly invoke lower level protocols like `ssh`. Signal handling and basic job control were included and standard I/O for the foreground process was routed to the user's terminal. In addition, `distsh` allowed process pipelines between remote processes. The design, however, routed all communications through the login host

creating a potential bottleneck. Significantly, at the time of publication, "remote" processes had only been created on the local host and not actually on a remote hosts. My attempts to contact the `distsh` authors have been unsuccessful.

Though full-featured Single System Image shells exist for specialized cluster operating systems and parallel commands can be executed through `pdsh` or scripts, there are no portable, general-purpose cluster shells providing SSI. Previous projects addressing this problem have not resulted in a production-ready shell and source code for those projects is unavailable.

# Chapter 2

# A GENERAL-PURPOSE SHELL FOR BEOWULF CLUSTERS

## 2.1 An Ideal Cluster Shell

Prior to design or coding, a number of desirable traits for a general-purpose cluster shell were identified and these traits represent an ideal against which `beosh` will be measured.

In support of a Single System Image (SSI), a cluster shell should transparently manage distribution of individual and pipelined commands and job control as if all processes are executing locally. A user should not need to know where in the cluster any particular process was executed. In addition to allowing command distribution across cluster nodes, it is also sometimes desirable to send the same command to multiple nodes for parallel execution, so a cluster shell should provide an option to broadcast a command to selected nodes. Any commands given by a user in a cluster shell should be executed with the same permissions the user would have if working through a local shell.

For SSI, a user should not be required to specify where any particular process

should execute, but some users may desire to launch processes on specific nodes so a cluster shell should allow users to specify the node for a process. Similarly, whether due to an administrative requirement that users exclusively reserve and use a subset of nodes or simply user desire, a cluster shell should allow specification of a subset of all cluster nodes to which it may assign user processes. However, because nodes may be used simultaneously by many users, a cluster shell should allow multiple users access to nodes simultaneously and provide security measures to protect users and their processes. Unless specific nodes are requested for specific processes, the shell should assign processes to lightly loaded nodes to make best use of available resources.

The ability to script a series of non-interactive commands is a useful and common single-system shell feature, so it should also be possible to create cluster shell scripts. In addition to basic scripting constructures like for loops, a cluster shell should provide a parallel construct like a forall loop allowing concurrent execution of loop commands on all selected nodes.

To reduce scalability issues, a cluster shell should avoid designs that contain any inherent bottlenecks such as routing all communications through any one node. Design should also be aware of limitations of commonly used utilities. Use of `rsh`, for example, which in turn uses privileged sockets (between 512 and 1023), limits the number of simultaneous connections to at most 256 and, hence, severely limits scalability. Potential alternatives might use Remote Procedure Calls (RPC), `ssh`, or basic sockets.

For portability and maintainability, a Beowulf cluster shell should not be restricted to a highly specialized operating system environment but should make use of standard system calls and utilities.

## 2.2 Project Goals: Providing a Foundation

For the `beosh` project, a set of initial goals was chosen to provide a solid foundation for realization of the ideal on Linux-based clusters or similar systems. These initial goals were chosen with the needs and priorities of the project's target clusters in mind, but without assumptions about the number of nodes in any cluster, any particular hardware, or any special operating system features. Any utility software used by `beosh` should be widely available so that `beosh` can be easily adapted to other clusters with Linux-like operating systems. The only initial cluster organization requirement is that the home file system must be shared by all nodes so that a valid command or path on one node would also be a valid on any other node and any input or output files will be accessible anywhere on the cluster. This is a standard setup for Beowulf clusters.

The central requirement for `beosh` is that it must provide a user with a Single System Image of the cluster by distributing commands and command pipelines across available cluster nodes or a subset of nodes without the user having to explicitly specify where any command should be executed. `Beosh` must also allow parallel execution of commands on a set of nodes as `pdsh` does. Because many users are already familiar

with `pdsh` syntax, `beosh` should use `pdsh` syntax for specifying session nodes.

As is common in single-system shells, `beosh` must provide a user with the ability to launch and manage multiple concurrent jobs by starting jobs in the foreground or background, killing or stopping the foreground job, and restarting a stopped job in the foreground or background in response to the same job control signals and commands used by single-system shells.

Users of the target clusters for this project are required to exclusively reserve and limit themselves to a subset of nodes through the Portable Batch System (PBS). To make `beosh` compatible with PBS, `beosh` should recognize when a user has reserved nodes through PBS by checking the `PBS_NODEFILE` environment variable and filtering out selected nodes not appearing in the reserved nodes file. Where other batch schedulers are used, an alternate environment variable pointing to a reserved nodes file could be specified.

# Chapter 3

# BEOSH DESIGN

## 3.1 Choosing a Project Approach

### 3.1.1 Potential Approaches

The two general types of commands a user is expected to execute are distributed commands and parallel commands. Pdsh is already in widespread use as a parallel command shell and it was desireable to make use of `pdsh` as the foundation for an enhanced application, as a pattern for the parallel aspects of a standalone application, or as a utility for a standalone application. Four potential approaches to `beosh` development were identified based on the relationship to `pdsh`.

1. Modify `pdsh` to add process distribution and job control support.

2. Create a new shell with process distribution and job control features and import parallel functionality from `pdsh`.

3. Create a new shell with process distribution and job control features and forward any parallel commands to `pdsh`.

4. Create a completely new system with all desired capabilities but with no consideration or use of existing software.

### 3.1.2 Plan A: Integrate With PDSH

The first plan for `beosh` was to start with `pdsh` code and modify it to support distribution of pipelined commands and job control. By starting with `pdsh`, `beosh` could take advantage of the routines and infrastructure of an already accepted and refined application. Parallel commands could be executed in the default way and daemons for distributed commands could easily be started in parallel. Node selection and filtering through a number of modules is also already built into `pdsh`.

As an exercise to become familiar with `pdsh` code, a `pdsh` module was written to filter nodes not currently reserved by the user through PBS[10]. It became clear with increasing familiarity with `pdsh` that its design is so rooted in parallel execution that it would be exceedingly difficult to modify it to support distributed commands or job control. Such extensive modifications would both negate many of the advantages of starting with an established application and likely inhibit its acceptance. On the other hand, if the existing basic design and functionality of `pdsh` were retained, few if any existing routines could be reused and so a completely independent code path would be needed for distributed command functionality. Another issue from a project maintenance standpoint complicates integration with `pdsh`. Integrating new functionality into `pdsh` or integrating `pdsh`-based parallel command code into a

new application would require either that the `pdsh` team take responsibility for the new code or that `beosh` managers update `pdsh`-based parallel code whenever `pdsh` is updated. For all of these reasons, it was decided that modification of `pdsh` was not a good option all and integration of `pdsh`-code into a new application would not be a viable alternative.

### 3.1.3 Plan B: A Separate Distributed Shell

The remaining options after ruling out integration with `pdsh` were to implement an entirely new standalone application with all desired functionality or to create a shell for distributed commands with job control that uses `pdsh` as a utility for executing parallel functions. Because `pdsh` is so widely used and is already a refined application for parallel command execution, there seemed to be no compelling reason to reinvent parallel command functionality. It was decided, therefore, that `beosh` would be a separate shell implementing distributed functionality, but it would require `pdsh` to be installed for parallel functionality and as a utility for hostname listing.

### 3.1.4 Language and Communications

When designing for a cluster system, fewer assumptions can be made about language and protocol support than for a typical single-user system. Because of the often streamlined and tightly controlled nature of clusters, it is important that dependencies for a highly portable cluster shell be as universal as possible. For these reasons, it was decided that the C language would be used as it is most likely to be supported on

any potential *nix cluster. Standard system calls with near-universal behavior should likewise be used.

For local and networked interprocess communication using C, there are a variety of options including remote procedure calls (RPC), remote sockets, and pipes. In addition to being most universal, sockets and half-duplex pipes allow for the most straightforward and direct connections between input, output, error, and administrative streams envisioned for `beosh` processes and job processes they manage.

For scalability, `beosh` design should avoid any potential bottlenecks such as routing communications through any one node or unnecessarily adding communications hops to data passing through pipelines. Use of sockets allows direct connections between job processes on different nodes without any unnecessary hops and allows construction of supporting management infrastructure across nodes without bottlenecks. Communication between parent processes and `fork()`-ed children through half-duplex pipes is likewise direct.

The client needs to be able to start backend command managers remotely as described in Section 3.2.1. Though it suffers from serious scalability limitations, `beosh` currently uses `rsh` because it allows for very straightforward client-side detection when remote processes exit and the ability to easily kill remote processes by killing child processes that have `exec()`-ed `rsh`. The scalability issues, however, make `rsh` an untenable long-term solution and an alternate design using `ssh` is described in Section 7.1.

## 3.2 Distributed Command Managers

### 3.2.1 General Backend Design

The initial system design envisioned for handling distributed commands was to have a complex, persistent, root-owned daemon on each node of the cluster to manage local slave shells on behalf of remote clients. For scalability and stability, persistent daemons would need to organize themselves into a network and handle the addition of new nodes and the loss of others dynamically with little or no hard-coded configuration. Some mechanism would also need to be established to restart any crashed daemons. In this system, it was thought that daemons could communicate their current loads with other daemons and the daemons would be responsible for distribution of jobs. Because daemons may handle sessions for many users, they would need some means to keep client sessions and jobs organized and protected from one another.

Under this design, all of the real power of `beosh` would be in the backend and the client would simply need to be able to attach to some daemon that would `fork()` a child with the user's permissions to initiate establishment of a pipeline with other daemons and pass output and error streams back to the client. While this plan has appeal, it seemed overly complicated and likely to suffer from frequent errors. Such a design would also impose constant overhead on cluster nodes. The organization of a self-assembled network of daemons would also be unlikely to fit arbitrary subsets of nodes requested by clients. Such a system might be practical for a very large cluster where the overhead of starting backend command managers ad hoc would

be prohibitively expensive and there is no intent to limit clients to subsets of nodes. Based based on expected use on the target clusters, however, this design was set aside in favor of running ad hoc backend distributed command managers (`dcmdmgr`s) to fit the subset of nodes in use by the client.

By starting backend command managers only as needed by a client session, there is no need for persistent, self-organizing daemons and each client session can have its own independent set of user-owned managers. Organization can be communicated to these `dcmdmgr`s to fit the needs of the client. With this straightforward option available, the complicated, all-powerful daemon approach was abandoned.

Top-level `dcmdmgr`s check at a configured interval to see if they have any children running; if not, the `dcmdmgr` exits. This idleness checking should prevent `dcmdmgr`s from running indefinitely if no client is making use of them while also allowing for long-running jobs.

By starting `dcmdmgr`s as user processes, permissions and environment variables such as the user's default shell should be the same for a `dcmdmgr` process as for the user. When started, a `dcmdmgr` receives as a required argument the port it should listen to. To reduce the risk of conflicts with `dcmdmgr`s started by other users, the port used by the client and all `dcmdmgr`s it starts is chosen based on the `pid` of the client. If EADDRINUSE is returned from a call to `bind()`, the port is not available. The `dcmdmgr` exits and when the client recognizes that it has exited, it can then select a new port and restart all `dcmdmgr`s with the new port until it finds one available for

all dcmdmgrs.

Dcmdmgrs fork() and exec() their commands using "/bin/bash -c command". This means most users should get the shell syntax and behavior they expect and it provides access to the scripting features of that single-system shell. If an alternate backend shell were desired, it could be made configurable.

For a dcmdmgr to learn its node's own local IP address, the hostname must appear in its local /etc/nodes file.

### 3.2.2 Job Management

There are many possible ways dcmdmgrs might handle the multiple jobs a session may be running concurrently on its selected nodes. Each node could have one dcmdmgr for each job distinguished by a unique port, but this would potentially require finding and tying up many free ports. Alternatively, if each session had one dcmdmgr on each node, the number of ports required could be kept low, but dcmdmgrs would then be required to keep multiple simultaneous jobs organized as well as their processes for appropriate handling of job control messages. Neither of these first two options was appealing.

To minimize port use and overhead from constant job ID checking in the backend, a client could start and connect to the tail dcmdmgr only, passing it all commands of the job pipeline and the nodes assigned to execute each command. The tail dcmdmgr would strip the tail command from the job and launch the next dcmdmgr, passing it the commands and nodes remaining in the job. This would continue until dcmdmgrs for

all commands in the job were launched. This design is highly scalable by avoiding any fanout issues from the client and allowing each job to have an independent pipeline, but other drawbacks prevented it from being chosen. All job control messages sent by the client would have to pass through the pipeline starting at the tail end and reflect back again when reaching the head for the client to get confirmation that the message was acknowledged by the whole job. While restart commands should begin with the tail process and end with the head process, stop and kill commands should begin with the head process and end with the tail. Stop and kill commands, then, would not begin to be carried out until they had been propagated the entire length of the pipeline. Establishing a pipeline in sequence this way also prevents multiple parts of the pipeline from being set up simultaneously.

A fourth option allows the client to initiate job control commands at either end of the pipeline and makes it possible to have multiple `dcmdmgr`s establishing connections simultaneously. The client remotely starts `dcmdmgr`s on all nodes and connects to each in turn from tail to head, passing each its command assignments. Dcmdmgrs `fork()` a child `dcmdmgr` to directly manage each command from the set of commands. Child managers for the job self-assemble into a bi-directional pipeline mirroring the job's process pipeline, connected to the client at each end. When all its required sockets have been connected, a `dcmdmgr` can `fork()` and `exec()` its assigned command. Each process `exec()`-ed by a `dcmdmgr` has its `stdin` and `stdout` directly connected to its upstream and downstream processes respectively. `Stderr` is connected to an

error handling thread of the parent `dcmdmgr` for forwarding to the client through the `dcmdmgr` pipeline. Each `dcmdmgr` retains sockets established for bi-directional job control administration and sockets to propagate error messages coming from its child process and from upstream `dcmdmgr`s.

The top-level `dcmdmgr` on each node does not accept a new set of commands until its children have finished establishing the pipeline for the current job. Although multiple pipelines can be running for a client at the same time on the same nodes, proper establishment of sockets between `dcmdmgr`s limits a client session to starting one pipeline at a time. This way, only one port is needed for each session, `dcmdmgr`s directly managing commands for a job never need to check session or job IDs so job control commands can be accepted and forwarded without additional overhead, and all input, output, and error traffic can be propagated through a job's pipeline as fast as the sockets will allow.

Minimal information is needed by `dcmdmgr`s once a job pipeline is established. Only the client needs to know if a job is running in the foreground or background. The backend `dcmdmgr`s only need to know if it should be running, stopped, or killed.

To prevent a malicious user from sending commands to a `dcmdmgr` started by a user with greater privileges, `dcmdmgr`s need to be able to verify that commands are comming only from the client that started them. Rather than implement an elaborate authentication scheme or incorporate any particular existing authentication mechanism, `beosh` leverages the fact that a port can only be in use by one process

on a node at a time. As long as the client is running on the original node and uses the same port to send command messages, a `dcmdmgr` can compare the node and port for every command connection against that of the original connection from the client. Assuming the legitimate client is always the first to contact any `dcmdmgr` it has started, `dcmdmgr`s can reject connections from illegitimate clients.

## 3.3 The Beosh Client

### 3.3.1 General Client Design

By giving the client responsibility for job distribution and for starting and monitoring the status of `dcmdmgr`s for its session, the client has more to manage than originally envisioned. However, with backend design completed and the decision to use `pdsh` for parallel command execution, design of the client is relatively straightforward and can closely follow that of a basic single-system shell.

Like `dcmdmgr`s, the `beosh` client runs as a user process giving the user the same permissions with `beosh` that they would have with a single-system shell. Tasks requiring higher privileges are handled by calling setuid processes such as `rsh` and `pdsh` from within `beosh` client.

### 3.3.2 Job Management and Job Control

A single-system shell `fork()`s and `exec()`s commands to protect itself and independent jobs from one another, keeping a handle on process group pids for job control in

a data structure like a linked list. While remote execution of jobs inherently reduces the threat of one job interfering with another, I/O errors will sometimes occur and `fork()`-ing a child to directly manage each job's I/O is still safer than having the top-level process or a thread directly manage jobs. The primary roles of the top-level `beosh` client process, like those of a single-system shell, are to parse user input and coordinate job control.

Unlike in a single-system shell, however, the children of the top-level client are not necessarily the intended recipients of user-given job control signals. The top-level client needs to be able to notify the child managing a target distributed job of the job control signal so that it can pass that message to the job. Two possibilities for notifying child clients were considered. First, the child could have its own signal handlers for the job control signals. The parent, then, could signal its children and have them respond according to their own signal handling routines. Signal handling, however, is a disruption to the normal flow of a process and it was felt that signaling children represented a threat to the operation of their primary I/O responsibilities. Implementing multiple routines to handle the same signals also increases the potential for confusion and error in maintaining the code. Finally, passing a simple message between processes seems to be a misuse of signals.

For these reasons, a pipe is created prior to `fork()`-ing child clients and this pipe is stored in the top-level client's job list. When a signal is received by the parent in distributed mode, a corresponding job control message is sent to the appropriate child

over its pipe. The child's administrative thread listens for parent commands on this pipe in addition to listening to its head and tail administrative sockets and sends the message down the appropriate socket. When the message has propagated through the pipeline and back to the client, the child notifies the waiting parent through a second pipe.

Job control is different in parallel mode, however, because child processes of the top-level client `exec() pdsh` to execute commands across selected nodes. The children, in this case, are the intended recipients of job control signals and the signal handling function, recognizing that `beosh` is in parallel mode, signals the target child directly.

A linked list of jobs is maintained by the top-level client with nodes for all current jobs. The client keeps a pointers to the foreground job node and the last stopped job node for targeting jobs when signaled by the user.

### 3.3.3 Job Distribution

When the top-level `beosh` client has parsed user input into jobs and jobs into commands, it sequentially `fork()`s a child for each job to distribute the commands of that job over the client's selected nodes. Though any number of jobs might be running simultaneously, jobs must be started sequentially because proper establishment of pipelines by backend `dcmdmgr`s requires that only one job be distributed at a time.

Unless the number of nodes available to users is effectively unlimited so that each process can execute on its own node, efficient use of cluster resources calls for some sort

of load balancing. If all processes induced identical load on nodes and communication carried no costs, simple round-robin assignment of commands would be as good as any other assignment approach. However, communication between nodes does tend to be significantly expensive and the burdens induced by processes vary widely.

Ideally, processes would be dynamically redistributed to balance load across all nodes. However, dynamic load balancing is challenging even for independent processes, let alone for connected processes distributed across nodes, and relocating processes carries significant overhead of its own. While `beosh` intends to consider load balancing, it is not the intent that the `beosh` project will solve the challenge of dynamic load balancing for highly-connected processes.

So, if processes will not be relocated once started, job distribution must be decided based on information that can be known before the job is started. Until a process is running, it is difficult to know anything about the load it will place on a node. The position of a process in a pipeline is also no indicator of load potential. While the current load on selected nodes can be known in advance, this requires some monitoring overhead and making effective use of this information is its own challenge.

Since there are no guarantees that using current load information would make load balancing more effective, it was decided to start with a simpler solution. Assuming that users of a cluster do so because they tend to run demanding processes, and further assuming that it is more beneficial to spread processes over as many nodes as possible than to minimize the number of connections between nodes, sim-

ple round-robin command assignment was considered first. Random distribution was also considered, but it fails to guarantee even distribution of processes or that all nodes will be utilized. Though round-robin assignment accomplishes both of these objectives, jobs may have more processes than the user has nodes. Because local communication is faster than communication over a network, a better distribution would divide job processes evenly over nodes, assigning consecutive processes to nodes that must execute more than one process. The `beosh` client combines this distribution technique with round-robin assignment of the starting node for each job.

Once the client has decided which commands will be assigned to each node, it can send `dcmdmgr`s on those nodes a job assignment with the upstream nodes for each command to enable the `dcmdmgr`s to assemble the job pipeline. Currently, the client distributes job assignments sequentially beginning with the `dcmdmgr` whose job assignment includes the tail process. This order is to improve the likelihood that the tail `dcmdmgr` will be ready to receive connections from the client when the client has finished distributing job assignments. This order also allows the client to retain the last socket made to the head `dcmdmgr` as an administrative socket for job control communication. Although parallel distribution of job assignments through multiple threads exists as a potential optimization, the average number of job assignments per job would have to be high enough to justify the added overhead of spawning threads and assumes simultaneous distribution of assignments over the network is even possible.

### 3.3.4 Dcmdmgr Management

In addition to the commandline parsing and job management roles of a single-system shell, the `beosh` client must be able to build a list of nodes on which it can distribute jobs and it must be able to manage the `dcmdmgr`s it starts on those nodes. Because `beosh` requires `pdsh` as a dependency for parallel command execution, the client is also able to make use of `pdsh` as a utility to build and filter the hostname list as specified through startup commandline options. In addition to hostname options as expected by `pdsh`, `beosh` adds two additional startup options, both of which affect the nodes on which `dcmdmgr`s will be started: one to specify parallel execution rather than the default distributed execution mode and one to override the PBS node filtering required on the target clusters. If parallel execution is specified, no `dcmdmgr`s will be started because `pdsh` doesn't use them. If PBS node filtering is enabled in either `beosh` or in `pdsh` through the `pbsnodefile` module, the override option turns off filtering.

Beosh supports filtering of nodes by current PBS node reservations either through the `beosh` client or through `pdsh` using the `pbsnodefile` module. The choice of whether to filter out unreserved nodes is made by setting `USE_PBS_FILTERING` to TRUE or FALSE in `beosh_list_node.h`. Likewise, if `pdsh` uses the `pbsnodefile` module, `PDSH_WITH_PBS_FILTERING` in `beosh_list_node.h` should be set appropriately.

The client is responsible to start `dcmdmgr`s on all nodes selected to receive com-

mands for a job and to restart `dcmdmgr`s that have timed out or otherwise exited since last used. It would have been possible to use `pdsh` to start `dcmdmgr`s on all selected nodes, but tracking them individually would have been difficult. So, instead, the client `fork()`s children to remotely start `dcmdmgr`s so that it can use their process IDs to recognize when `dcmdmgr`s exit. For convenience in development, child clients `exec() rsh` to start `dcmdmgr`s. However, as has been noted already, `rsh` draws from the limited set of reserved ports and, therefore, has a serious scalability problem. Use of `rsh` should be replaced by `ssh` or another less restricted remote protocol before `beosh` can be considered ready for large-scale cluster environments. The choice of remote protocols could be made configurable in the future.

Because all `dcmdmgr`s for a session share a common port number, a session can only start one `dcmdmgr` on any node. There may be cases, however, where a user wants to specify that a particular node should be utilized more heavily than others. It is not uncommon, for example, to have some nodes with more or faster processors or memory than others. Although it is not yet implemented in `beosh`, it should be a simple matter to recognize when a node is specified more than once at startup and to assign more commands to that node accordingly.

Originally, port selection began with each user's `uid`, but later used the client `pid` instead. A weakness of basing port selection on a `uid` is that every client started by the same user will start in the same place to find a port. If a user has multiple clients open at once and many ports are tied up, it is possible that they will exceed

their configured number of retries before getting past the block of used ports. If `uid`s are assigned consecutively on a system, multiple users could likewise find themselves frequently competing for ports. Though less prone to confict than `uid`, `pid`s are still assigned consecutively on a system and the potential for conflicts remains fairly high. The client now uses a hash function based on its `pid` to separate ports even between clients with consecutive `pid`s.

Currently, the client starts `dcmdmgr`s on all its nodes during startup and refreshes all `dcmdmgr`s prior to distributing each job. The advantage of this arrangement is that it minimizes the chance that a `dcmdmgr` will exit between the time it is checked and the time the client tries to send it a job assignment. Unfortunately, this puts overhead where the user is most likely to notice it. Two alternative approaches might reduce the user's awareness of `dcmdmgr` management overhead, but care would need to be taken to keep the risk of sending job assignments to unprepared `dcmdmgr`s low.

If the client used a single `dcmdmgr`-managing thread, it could be continually monitoring and refreshing `dcmdmgr`s so that they are prepared when a job is received. To prevent the managing thread from changing the port number and restarting `dcmdmgr`s with the new port at the same time the main thread is trying to send job assignments, a mutex locking the `dcmdmgr` list would be needed.

It would also be possible to have a thread for each `dcmdmgr` which would allow the client to start and refresh all `dcmdmgr`s in parallel. It might be more difficult, though, to recognize when a `dcmdmgr` can't start with the current port and all `dcmdmgr`s need

to be restarted with a new port and coordinate the transition of all `dcmdmgr`s to the new port. As with the single thread, a mutex would need to protect simultaneous use of the node list by the main client thread and `dcmdmgr`-managing threads. Because each thread would manage a different `dcmdmgr`, there would be no conflicts between managing threads as long as each is notified of port changes, but with each thread contending with the main thread for access to the node list, they would all be forced to contend for the sole mutex. This option seems less likely to yield a real performance improvement than the single thread option.

When the client recognizes that a `dcmdmgr` has exited, an attempt is made to start a new one on the node with the same port number. This action should account for any `dcmdmgr`s that time out after sitting idle beyond their configured wait time. If the new `dcmdmgr` fails to start, the client assumes the port is in conflict and advances the port number. All `dcmdmgr`s need to use the same port, so when the port is changed, all `dcmdmgr`s are killed and new ones are started with the new port number. This approach, however, fails to account for the significant problem of a `dcmdmgr` that exits in error during pipeline establishment for a job.

If a `dcmdmgr` exits in error while involved in pipeline establishment, it leaves other `dcmdmgr`s in a hung state. Dcmdmgrs that are not yet fully connected will wait indefinitely for connections that will never come. Child job processes of fully connected `dcmdmgr`s will wait indefinitely for input that will never come. The child client process managing the job will wait indefinitely for the pipeline to be established. When

the top-level client refreshes the exited `dcmdmgr`, it does not recognize that the set of `dcmdmgr`s is in an unusable state. The job that failed to start is lost and subsequent job assignments sent to hung `dcmdmgr`s will not be received as job assignments because those `dcmdmgr`s expect only connections for pipeline establishment.

Unfortunately, the client cannot differentiate between a `dcmdmgr` that has timed out and one that has exited in error because it detects exit of `dcmdmgr`s indirectly through its child processes that remotely executed the `dcmdmgr`s. To maintain a complete set of `dcmdmgr`s capable of receiving jobs regardless of exit conditions, the client restarts all top-level `dcmdmgr`s whenever any of them exits. This is a straightforward solution that seems to account for errors of many kinds, but it could result in frequent unnecessary restart overhead for users who do not make continuous use of all of their available nodes. To minimize the frequency of unnecessary restarts, the timeout period for `dcmdmgr`s should be set high enough to cause only rare interruptions to active user sessions while ensuring that truly idle sessions do not tie up ports indefinitely.

On exit, the client kills any of its remaining `dcmdmgr`s to leave a clean system.

To coordinate the current working directory of the client and all `dcmdmgr`s, the client connects to any `dcmdmgr` it has started and notifies it of its current working directory so it can change its own current working directories to match. Likewise, whenever the client's current working directory changes, it contacts all `dcmdmgr`s to notify them of the change.

### 3.3.5 Socket Management

For each job, the child client managing the job must manage:

1. the upstream (tail) and downstream (head) administrative sockets for passing job control messages to `dcmdmgr`s,
2. the input socket to receive the output of the job pipeline,
3. the input socket to receive error messages from the job,
4. and a pair of pipes to the parent client from which it receives and confirms job control directives.

A child client managing a job handles socket I/O with an administrative thread and an error thread in nearly identical fashion to `dcmdmgr`s while the main thread handles reading the output of the job pipeline.

The administrative thread listens for job control directives from the parent client and propagates them down the appropriate upstream or downstream administrative socket. It also listens for returned directives that have been propagated through the administrative loop. When a response is received, the child notifies the waiting parent through a second pipe.

The error thread listens for error messages from the job's error pipeline and writes them to `stderr`. When it receives EOF from the socket, it recognizes the job as completed and exits.

When the main thread, listening for output from the job, receives EOF, it recognizes the job as complete and waits for the error thread to join before exiting. Waiting for the error thread accounts for the less direct route error messages must take as compared to output and allows any lagging error messages to be received before destroying the error thread.

### 3.3.6 I/O Redirection

Input redirection is handled by the `dcmdmgr` managing the head process of a job. Output and error redirection are the responsibility of the client. Output and error redirection apply only to the job with which they are associated. The client's own error messages are always written to `stderr`.

### 3.3.7 Security

Beosh security is limited because security is not the primary goal of the `beosh` project and time constraints restricted development to only core features.

The target clusters are separated from outside networks by firewalls and all commands requested by users are executed with the user's own permissions. To prevent malicious users from submitting commands to `dcmdmgr`s of other users, `dcmdmgr`s accept command connections only from the first node and port that connects to them on the assumption that the first connection will be from the legitimate client.

All socket traffic is unencrypted, however, and users will need to be made aware of the risks present in using `beosh` as is. To allow for future security modifications,

`beosh` should avoid design choices that will prevent use of encryption or more secure authentication through Kerberos, `ssh`, or some other means later on.

# Chapter 4

# OPERATION OF DISTRIBUTED COMMAND MANAGERS

## 4.1 Assembling a Pipeline

When the user enters a job, the client parses it and selects nodes over which to distribute commands as it sees fit. If not running already, the client remotely starts a `dcmdmgr` on each selected node, passing each the common port number shared by all for establishing sockets (Figure 4.1). The protocol used to start `dcmdmgr`s (`rsh`, `ssh`, etc.) is up to the client. Choosing a port is also the client's responsibility. If the specified port is already in use, the `dcmdmgr` exits. It is up to the client to recognize and handle this. If the `dcmdmgr` is able to start, it listens on its assigned port for a command assignment.

The client connects to each `dcmdmgr` in turn from tail to head and passes it its job assignment (Figure 4.2). The job assignment includes:

1. The number of commands in the job that the `dcmdmgr` will be managing.
2. The set of commands it will manage for this job. Each command assignment includes:

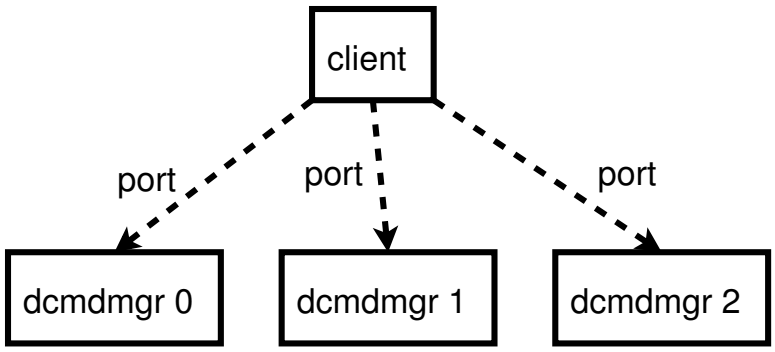


Figure 4.1. The client starts one `dcmdmgr` on each selected node.

(a) The command string.

(b) The input file. This should be empty string for all but head `dcmdmgr` and is ignored by all but head `dcmdmgr`.

(c) The node responsible for the next command upstream from this one in the pipeline. This is used to establish connections to that node's `dcmdmgr`. The"upstream" node may be the current node. If node is NULL, this command is the head of the pipeline.

There is no need for `dcmdmgr`s to know a job ID or anything of the sort. Keeping jobs straight is the responsibility of the client. The client is responsible to send each `dcmdmgr` its commands in tail to head order so the `dcmdmgr`s can assemble the pipeline correctly.

The first socket from the client is assigned to the `dcmdmgr`'s upstream administrative socket file descriptor. All `dcmdmgr`s except the `dcmdmgr` handling the head com-

mand will eventually replace this socket with one from its actual upstream `dcmdmgr`.

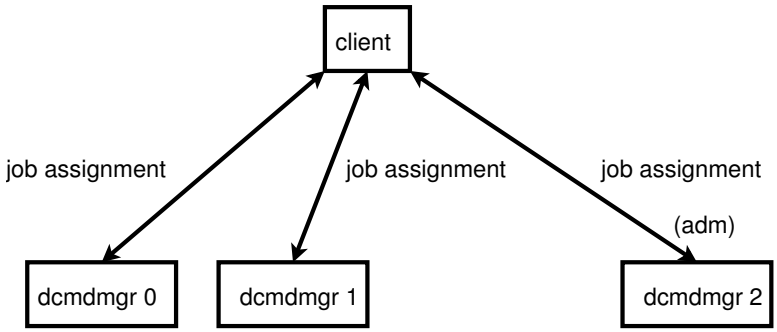


Figure 4.2.: The client sends each `dcmdmgr` its job assignment. Assignment order is from tail to head with the head `dcmdmgr` retaining the socket as its upstream administrative socket.

Each `dcmdmgr` handles its commands in the order received, listening for new socket connections from the downstream command's `dcmdmgr` or, for the tail command, the client (Figure 4.3). The client, as the process furthest downstream, initiates the assembling of the pipeline by contacting the `dcmdmgr` that was assigned the tail command (Figure 4.4). This first `dcmdmgr` listens for three socket connections and assigns them to the following file descriptors in order:

1. Downstream administrative socket (open for 2-way communication)

2. Downstream error (open for writing only)

3. Downstream output (open for writing only)

The downstream-connected `dcmdmgr` now `fork()`s a child to handle all additional management for the current command (Figure 4.5). The parent `dcmdmgr` closes its

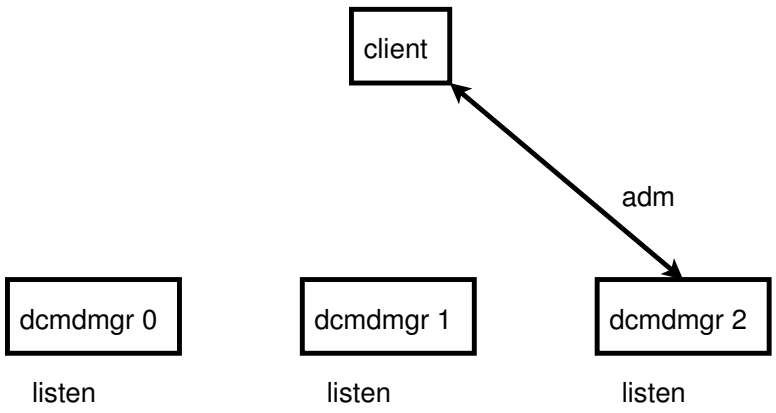


Figure 4.3.: All `dcmdmgr`s listen for administrative, error, and output socket connections from downstream.

handles on the downstream sockets and starts listening for downstream connections for the next command in its job assignment.

When all of its commands are being handled by a child `dcmdmgr`, the parent `dcmdmgr` closes all of its open socket file descriptors and returns to waiting for additional job assignments from clients.

If it has one, a newly `fork()`-ed child `dcmdmgr` contacts its upstream `dcmdmgr`, which may even be the parent `dcmdmgr` on the same node, to establish sockets for the following 3 file descriptors in order:

1. Upstream administrative socket (open for 2-way communication)

2. Upstream error (open for reading only)

3. Upstream input (open for reading only)

The other ends of these sockets are being assigned to the downstream set of file

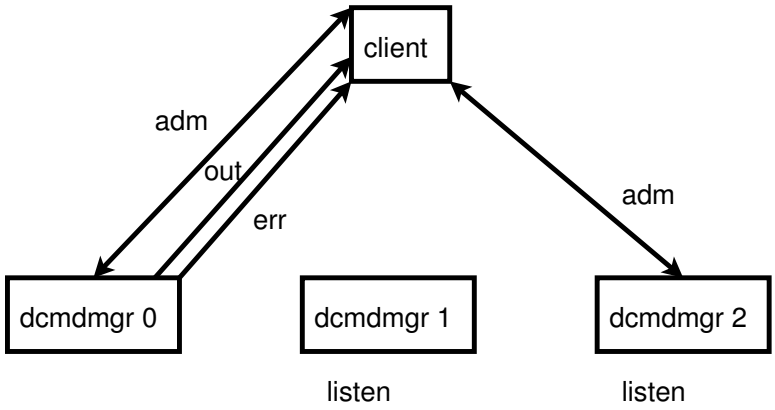


Figure 4.4.: As the furthest downstream process, the client initiates pipeline establishment by connecting to the tail `dcmdmgr`.

descriptors by the contacted `dcmdmgr` (Figure 4.5).

If the child `dcmdmgr` has no upstream `dcmdmgr` node for its command, it is the head of the pipeline and has all of the connections it needs.

Once all of its communications connections have been made, the `dcmdmgr` creates its pipe for `stderr` and `fork()`s and `exec()`s its command (Figure 4.6). Because the pipeline is established from tail to head, there is no need to wait for any additional go-ahead confirmation.

After `fork()`-ing, the new command child sets its `stdout` to the downstream output file descriptor, its `stderr` to the new error file descriptor, and its `stdin` to the upstream input file descriptor. The parent `dcmdmgr` closes its handles on the downstream output and upstream input file descriptors.

When all `dcmdmgr`s have `fork()`-ed and `exec()`-ed their commands, the job pro-

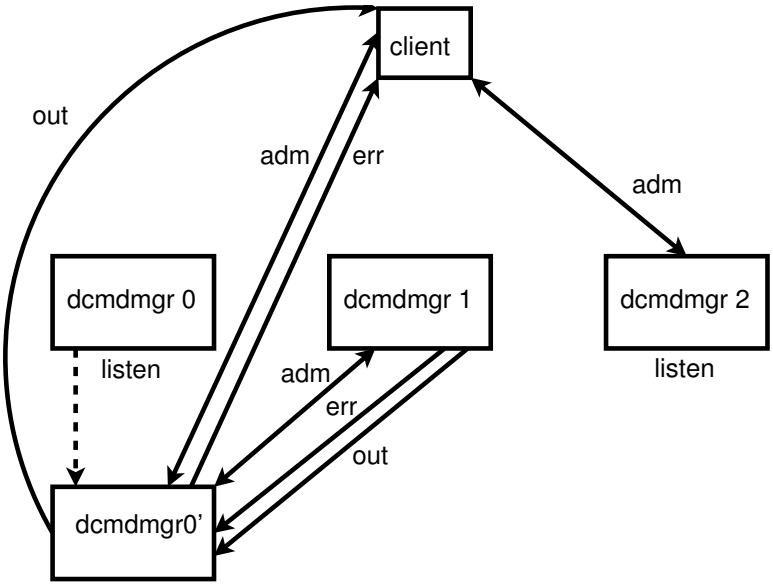


Figure 4.5.: The downstream-connected tail `dcmdmgr` `fork()`s a child to contact its upstream `dcmdmgr` which may be the parent or a `dcmdmgr` on another node.

cess pipeline is complete, a parallel pipeline of sockets for error propagation is established, and a complete, 2-way ring of administrative sockets has been formed joining all `dcmdmgr`s and the client (Figure 4.7, Figure 4.8).

## 4.2 Shutting Down a Pipeline on Completion

When a `dcmdmgr`'s child has finished as detected by a call to `waitpid()`, it sends a command completed message to its downstream `dcmdmgr` over its downstream administrative socket. Any `dcmdmgr` that has received a command completed message from its upstream `dcmdmgr` is free to exit. When the client receives a command completed message from the tail `dcmdmgr`, it knows the job is finished, but to complete

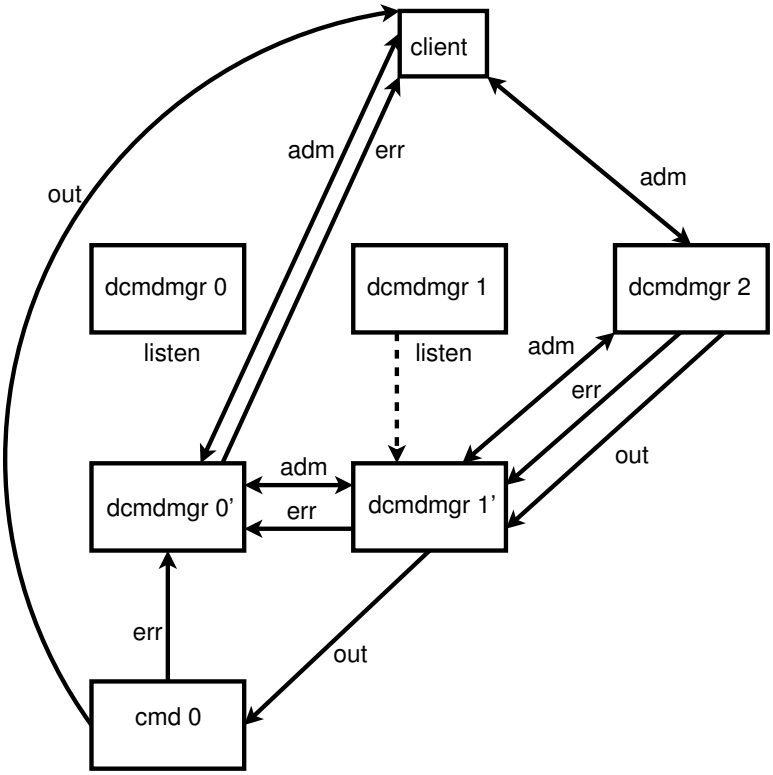


Figure 4.6.: The fully connected tail `dcmdmgr` creates a pipe to receive error messages from its child. When the command is `fork()`-ed and `exec()`-ed, its `stdout`, `stdin`, and `stderr` are assigned to the sockets and pipe established by its parent. While this is happening, the second `dcmdmgr` is establishing connections to its upstream `dcmdmgr`.

the cleanup of the pipeline, it must send a command completed message to the head `dcmdmgr`. The reason for delaying the shutdown of the pipeline is for the sake of error messages that may be propagating through the pipeline more slowly than process output.

If a `dcmdmgr` has sent its command completed message but has not received a command completed message from its upstream process for some timeout interval,

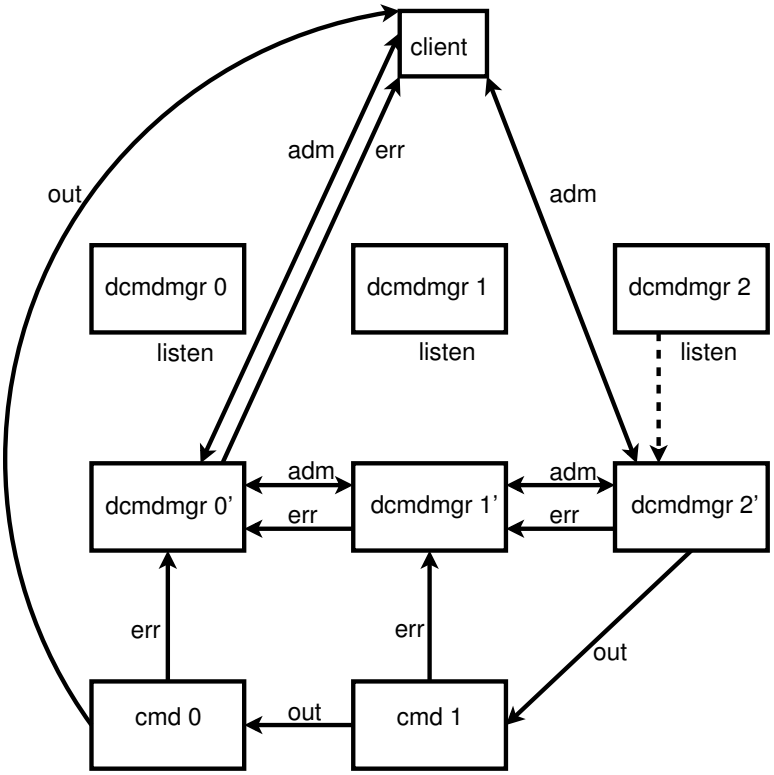


Figure 4.7.: The second `dcmdmgr` `fork()`s and `exec()`s its command. The head `dcmdmgr` has no upstream `dcmdmgr` and already has its "upstream" administrative socket. It is fully connected and ready to `fork()` and `exec()` its own command.

it should be free to exit. This could happen, for example, to the head node if the client dies before it is able to send the finished head node its command completed message. However, there is currently no timeout to account for a hung `dcmdmgr`. This has not actually been a problem because the call to `select()` in the administrative thread returns when a socket connection is broken. When the administrative thread recognizes that one of its sockets is broken, it knows it is of no more use and sends an error notification down its remaining socket before exiting, leaving the main thread

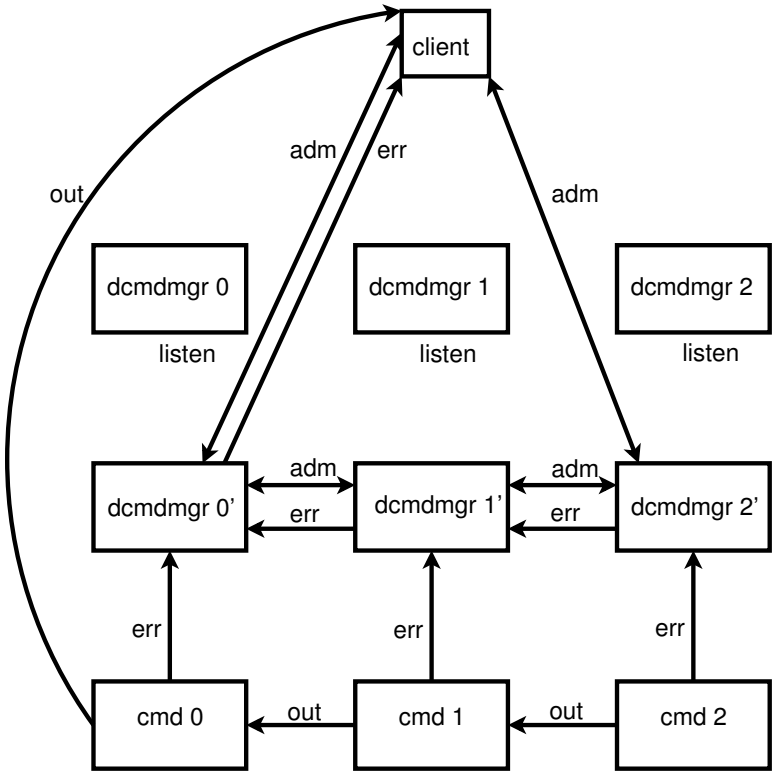


Figure 4.8.: When the head `dcmdmgr` has `fork()`-ed and `exec()`-ed its command, the process pipeline and parallel `dcmdmgr` pipelines are fully connected and operational. All top-level `dcmdmgr`s are free to listen for their next job assignments.

free to exit.

## 4.3 I/O Threads and Job Control

After all upstream and downstream socket connections have been established and its assigned command has been `exec()`-ed, a `dcmdmgr` creates a thread for each of the following tasks:

1. Reading from the upstream and downstream administrative sockets, handling

messages received on those sockets, and propagating received messages from one administrative socket to the other.

2. Reading from the upstream error socket and child error pipe and propagating received messages to the downstream error socket.

To avoid busy-waiting, I/O threads call `select()` to block indefinitely until one of the thread's assigned file descriptors is readable.

The ring of administrative sockets allows the client to send a job control message through the pipeline starting at either end. When `select()` returns in the adminThread, the upstream administrative socket is checked first for a kill command followed by a stop command and finally a command completed notification. If none of these commands has been received, the downstream administrative socket is checked for a restart command.

To stop a job, the client sends a stop message to the head `dcmdmgr`. The head `dcmdmgr` stops its child with a `SIGTSTP` signal before passing the stop message to its downstream `dcmdmgr` over its downstream administrative socket to prevent a downstream command from being stopped before an upstream command. Each `dcmdmgr` receiving the stop message likewise stops its child and propagates the message downstream. When the client receives the stop message from the tail `dcmdmgr`, it knows the job is stopped.

Killing a job works similarly to stopping a job. The client sends a kill message to the head `dcmdmgr` and it is propagated downstream. However, when killing a job,

each `dcmdmgr` propagates the kill message before killing its child with `SIGINT`. This order is to avoid having the main `dcmdmgr` thread recognize the exit of the child and exit itself, thereby terminating all related threads, before the adminThread is able to propagate the kill command. When the client receives the kill message from the tail `dcmdmgr`, it knows the job is dead.

To resume a job, the client sends a resume message to the tail `dcmdmgr`. The tail `dcmdmgr` restarts its child with a `SIGCONT` signal and sends the resume message to its upstream `dcmdmgr` over its upstream administrative socket. Each `dcmdmgr` receiving the resume message resumes its child before propagating the message upstream. This order is to prevent an upstream command from restarting before its downstream command. When the client receives the resume message from the head `dcmdmgr`, it knows the job is running again. Note that it is completely irrelevant to a `dcmdmgr` if its child process is running in the foreground or the background. Foreground and background only have meaning to the client.

When `select()` returns in the errThread, first the upstream error socket is checked followed by the `exec()`-ed command's error descriptor. Priority, then is given to error messages being propagated from upstream. However, when either error file descriptor is read, reading stops at the end of a line or at the specified buffer size and `select()` is called again. This may result in interleaving of error streams, but it is intended to prevent downstream error messages from being permanently suppressed by a prolific upstream process.

# Chapter 5

# OPERATION OF THE BEOSH CLIENT

## 5.1 The Main Client Loop

On startup, the command line options are parsed. Host selection options are expanded by `fork()`-ing and `exec()`-ing `pdsh` to execute the "hostname" command with all node selection options from the commandline. `Awk` is used to parse only the hostnames out of the `pdsh` output. If `pdsh` was not configured to use the `pbsnodefile` module, `beosh` was configured to require PBS node filtering, and the user did not specify the override option, the node list returned by `pdsh` is filtered by the user's currently reserved PBS nodes. If errors occur during parsing, `beosh` client exits with appropriate messages of woe or chastisement.

The client gets user information including user name, user ID, and home directory with with `getpwuid()`. If using `rsh` for starting `dcmdmgr`s, permission to execute remote commands is verified with `ruserok()`.

`Dcmdmgr`s are started on all selected nodes using `rsh`. The port number sent to `dcmdmgr`s is selected based on the client's `pid`. Ports are limited to the range 5000 to 49125 to avoid Berkley ephemeral ports and "correct" ephemeral ports. Port selection

starts with 5000 + `pid` and increments from there, wrapping back to 5000 if reaching the upper bound. Whenever `dcmdmgr`s are started or refreshed, if they repeatedly exit, the port is assumed to be in conflict and a new port is chosen by incrementing the previous port number. All `dcmdmgr`s are restarted when a new port is chosen. In any given round of `dcmdmgr` refreshing, if a configured maximum number of ports has been tried without success, the client exits.

To manage `dcmdmgr`s, the client `fork()`s a child for each node and stores its pid in the node's node in the node list. Each child process `exec()`s `rsh` to start a `dcmdmgr` on its node passing it the session port as a commandline argument. Whenever the client needs to refresh `dcmdmgr`s, it can iterate over the node list and check for children that have exited with `waitpid()`, starting new `dcmdmgr`s as needed.

Signal handling for `SIGINT` and `SIGTSTP` is set to notify the client child managing the current foreground job of the signals if in distributed mode. If in parallel mode, signals will be passed directly to the foreground `pdsh` process.

If a trailing command was given at the commandline, `beosh` runs noninteractively and exits after executing only the given command. Otherwise, `beosh` runs interactively, accepting input indefinitely with `readline()` until the user enters the "`exit`" command. As commands are entered, they are added to readline history for convenience.

The client initializes signal handling for `SIGINT ånd SIGTSTP` to call a method that will propagate those signals to the current foreground process.

User input is parsed to break each commandline into pipelines (jobs) and then further into commands. Input, output, and error redirection files are stored in command structures with their associated commands. If a job has been specified as a background job, that information is also stored in the job structure. The command line structure containing its parsed parts is sent to the `execJobs()` method which iterates over the jobs in the structure.

## 5.2 Job Execution

When `execJobs()` receives a command line structure from the main program loop, it iterates over the jobs, handling them sequentially. If the current job has one command and that command is a built-in `beosh` command, it is handled by the `beoshBuiltIn()` method. Built-in commands include the job control directives "`fg`" and "`bg`", the job status command "`jobs`", the change directory command "`cd`", the readline history command "`history`", and the exit commands "`exit`" and "`logout`". When a built-in command has been handled, `execJobs()` moves to the next job in the command line structure.

If the job is not a built-in command, a pipe is created for communicating job control messages to the soon-to-be-`fork()`-ed child. In parallel mode, child processes are signaled directly, so no pipe is needed, but the parent expects all of its children to have one for populating its process list. If the client is in the default distributed mode, `dcmdmgr`s are refreshed and the pointer to the starting node is initialized if necessary.

A child is `fork()`-ed to be the dedicated manager of the job if in distributed mode and to be `exec()`-ed as a `pdsh` process if in parallel mode. The top-level client stores the child pid, job status, and job control pipe for each child.

A child client, immediately after being `fork()`-ed, sets its signal handling to defaults so that it will not compete with the parent for signal handling and creates a new process group for itself. In distributed mode, the child calls `dcmdExec()` to coordinate establishment of a `dcmdmgr` job pipeline and threads to handle I/O responsibilities of the client for the job. In parallel mode, the child calls `pdshExec()` to forward the job to `pdsh`.

Each job is assigned the next available job number. This is 1 if there are no other unfinished jobs or 1 plus the highest unfinished job number.

When all jobs have been handled in `execJobs()` and the function returns, the parent client clears the command line structure, identifies, reports, and clears any completed background jobs, and waits for the next command line.

### 5.2.1 Distributed Jobs

The child client breaks its job into command assignments for distribution over the selected nodes, beginning with the next node in the node list after the last node given a command assignment in the previous job. This round-robin adjustment to the starting node ensures that commands will not be started on a few nodes repeatedly while other available nodes never receive an assignment.

Load balancing is simple, but serves to distribute commands evenly over available

nodes while also minimizing network traffic. First, the minimum number of commands that will be be assigned to any node is calculated with integer division as `commands / nodes`. Then, the number of remaining commands is calculated with modular division as `commands % nodes`. When assigning commands, each node will receive the minimum number of consecutive commands from the job (which may be none) and, if its index in the assignment loop is less than the number of remainder commands, it receives one more.

The child client connects to the `dcmdmgr` on each node in turn beginning with the node that will handle the tail of the pipeline and ending with the node that will handle the head of the pipeline, passing each `dcmdmgr` its commands in tail-to-head order. The socket connection to the head `dcmdmgr` is retained by the client as the client's head administrative socket and by the head `dcmdmgr` as its "upstream" administrative socket.

Each command assignment includes the command string and its arguments, the input file name if input redirection was specified and the command is the head command, and the name of the node to which the upstream command's `dcmdmgr` is assigned. The upstream `dcmdmgr` may actually be the same one assigned the current command. Fear not. The `dcmdmgr`s can handle it.

When all commands have been distributed, if output redirection was specified for the job, the client opens the output file and `dup()`s its `stdout` to the file. Error redirection is handled in the same way.

The client, as the furthest downstream process of the job pipeline, initiates establishment of the pipeline connections by contacting the tail `dcmdmgr` to open, in order:

1. An administrative socket,
2. An error socket,
3. And an input socket.

The client's input socket receives the final `stdout` of the pipeline and the error socket receives the final `stderr` of the pipeline. Once the client has made these connections to the tail `dcmdmgr`, the `dcmdmgr`s handle the rest of the connections and start their commands.

To handle the pipeline error stream and administration sockets for job control, the child client starts two threads: an error thread and an administrative thread. The main client thread reads pipeline output directly. The error thread simply handles input received from the error socket. The admin thread handles I/O from both the head and tail administrative sockets and also propagates signals received from the parent client.

Once the error and admin threads are started, the main child client thread reads from its input socket until EOF is reached. When EOF is read, the child client closes its input and head administrative socket connections, waits for the error thread to finish, and then closes the error and tail administrative sockets before exiting.

With this organization, only the parent process is aware of which jobs are in the foreground or background. The child clients need only propagate job control commands from the parent and the `dcmdmgr`s likewise need only propagate signals and respond to propagated signals.

### 5.2.2 Parallel Jobs

In parallel mode, the child client `exec()`s `pdsh` with the node selection options originally given to `beosh` at startup. The parent client, as in distributed mode, creates a process node for the job for use in job control.

# Chapter 6

# EXAMPLE USAGE

```
[mvail@rookery ~]$ qstat -a
[mvail@rookery ~]$ pdsh -a hostname
node0: rookery.boisestate.edu
node4: node4
node1: node1
node2: node2
node5: node5
node6: node6
node7: node7
pdsh@rookery: node3: connect: No route to node
[mvail@rookery ~]$ beosh -a
Reserve nodes through PBS and try again.

[mvail@rookery ~]$ pbsget -6

########################################################################
Allocate cluster nodes via PBS for running interactive parallel jobs.
########################################################################

Trying for 6 nodes

qsub: waiting for job 48.rookery.boisestate.edu to start
qsub: job 48.rookery.boisestate.edu ready

[mvail@rookery PBS ~]:qstat -n

rookery.boisestate.edu:
                                    Req'd   Elap
Job ID     Username Queue    NDS Time  S Time
---------- -------- -------- --- ----- - -----
```

```
48.rookery mvail    default  7   00:30 R   --
   node0/0+node7/0+node6/0+node5/0+node4/0+node2/0+node1/0
[mvail@rookery PBS ~]:beosh -a
pdsh@rookery: node3: connect: No route to node
beosh> pmode
beosh> hostname
node0: rookery.boisestate.edu
node4: node4
node1: node1
node5: node5
node2: node2
node6: node6
node7: node7
pdsh@rookery: node3: connect: No route to node
beosh> dmode
beosh> status

BEOSH has been running for 47 seconds.

Distributed Mode - Commands are distributed evenly across nodes.
Current port: 33598

Selected Nodes:
node0
node2
node1
node4
node7
node5
node6

Nodes were filtered by PBS.

beosh> sleep 10 &

[1] 28668

beosh> sleep 15 &

[2] 28675

beosh> sleep 10 &
```

```
[3] 28678

beosh> sleep 20 &

[4] 28681

[1] Done sleep 10  &
beosh> sleep 5 &

[5] 28686

beosh>
[3] Done sleep 10  &
beosh>
beosh>
[2] Done sleep 15  &
beosh>
[5] Done sleep 5  &
beosh> jobs
[4] Running sleep 20  &
beosh>
beosh>
beosh> jobs
[4] Running sleep 20  &
beosh>
beosh>
[4] Done sleep 20  &
beosh> jobs
beosh> pwd
/home/mvail
beosh> cd ..
beosh> pwd
/home
beosh> cd
beosh> pwd
/home/mvail
beosh> ls -alh
total 1.9M
drwxr-xr-x   8 mvail mvail 4.0K Jul  4 16:18 .
drwxr-xr-x  13 root  root  4.0K Jun  7 10:03 ..
-rw-------   1 mvail mvail  17K Jun 21 17:18 .bash_history
```

```
-rw-r--r--   1 mvail mvail   24 May  1 17:49 .bash_logout
-rw-r--r--   1 mvail mvail  191 May  1 17:49 .bash_profile
-rw-r--r--   1 mvail mvail  124 May  1 17:49 .bashrc
-rw-r--r--   1 mvail mvail 5.5K May  1 17:49 .canna
-rw-r--r--   1 mvail mvail  438 May  1 17:49 .emacs
-rw-r--r--   1 mvail mvail  120 May  1 17:49 .gtkrc
drwxr-xr-x   3 mvail mvail 4.0K May  1 17:49 .kde
drwx------   2 mvail mvail 4.0K Jun  7 16:09 .ssh
drwxrwxr-x   3 mvail mvail 4.0K Jun  1 10:06 .subversion
-rw-------   1 mvail mvail 7.0K Jul  4 16:18 .viminfo
-rw-r--r--   1 mvail mvail  658 May  1 17:49 .zshrc
lrwxrwxrwx   1 mvail mvail   23 Jun  4 18:58 beosh -> dcmdmgr/src/
beosh/beosh
-rw-r--r--   1 mvail mvail  636 Jun 21 16:03 beosh_snapshot_06_21.
README
-rw-rw-r--   1 mvail mvail 1.6M Jun 21 17:12 beosh_snapshot_06_21.tgz
-rw-r--r--   1 mvail mvail  929 Jul  4 16:23 dcmd.log
drwxrwxr-x   7 mvail mvail 4.0K Jun 21 15:21 dcmdmgr
-rwxr-xr-x   1 mvail mvail  490 Jun 21 15:26 make_beosh.sh
-rwxr-xr-x   1 mvail mvail  487 Jun 21 15:27 my_make_beosh.sh
drwxrwxr-x   2 mvail mvail 4.0K Jun 21 15:41 projectbackups
drwxrwxr-x   3 mvail mvail 4.0K May  1 17:51 pvm3
-rw-rw-r--   1 mvail mvail 223K Jun 21 17:11 src.tgz
beosh>
beosh> ls -alhR | grep mvail | grep rw | grep Jun | sort | sort -r
 | grep K | grep _ | grep c$ | grep beosh
-rw-rw-r--  1 mvail mvail 9.2K Jun  4 16:22 beosh_list_node.c
-rw-rw-r--  1 mvail mvail 8.9K Jun 15 22:03 beosh_parse.c
-rw-rw-r--  1 mvail mvail 3.7K Jun  4 16:22 beosh_list.c
-rw-rw-r--  1 mvail mvail 3.2K Jun  7 19:04 beosh_error.c
-rw-rw-r--  1 mvail mvail 2.6K Jun  4 16:22 beosh_node.c
-rw-rw-r--  1 mvail mvail 1.9K Jun  4 16:22 beosh_node_node.c
-rw-rw-r--  1 mvail mvail  19K Jun 21 15:41 beosh_util.c
-rw-rw-r--  1 mvail mvail  19K Jun 17 17:08 beosh_builtin.c
-rw-rw-r--  1 mvail mvail  16K Jun 21 15:41 beosh_io.c
-rw-rw-r--  1 mvail mvail  16K Jun 17 17:08 beosh_exec.c
beosh> help

BEOSH BUILT-IN COMMANDS:

'help'
        - display this message
```

```
'exit' or 'logout'
        - quit beosh

'cd [path]'
        - change directory if in distributed mode

'jobs'
        - list current status of stopped or background jobs

'fg [job number]'
        - run last stopped job or numbered background job in the
          foreground

'bg [job number]'
        - run last stopped job or numbered background job in the
          background

'history [n]'
        - show all commands entered in the session or only the last
          n commands

'dmode'
        - switch to distributed mode

'pmode'
        - switch to parallel mode

'status'
        - display information about current beosh settings and nodes

'version'
        - display current beosh version

beosh> status

BEOSH has been running for 256 seconds.

Distributed Mode - Commands are distributed evenly across nodes.
Current port: 33598

Selected Nodes:
```

```
node0
node2
node1
node4
node7
node5
node6

Nodes were filtered by PBS.

beosh> hostname
rookery.boisestate.edu
beosh> hostname
node2
beosh> hostname
node1
beosh> hostname
node4
beosh> hostname
node7
beosh> hostname
node5
beosh> hostname
node6
beosh> hostname
rookery.boisestate.edu
beosh> hostname
node2
beosh> hostname
node1
beosh> hostname
node4
beosh> hostname
node7
beosh> hostname
node5
beosh> hostname
node6
beosh> hostname
rookery.boisestate.edu
beosh> pmode
beosh> hostname
```

```
node0: rookery.boisestate.edu
node1: node1
node6: node6
node5: node5
node4: node4
node2: node2
node7: node7
pdsh@rookery: node3: connect: No route to node
beosh> status

BEOSH has been running for 300 seconds.

Parallel Mode - Commands are forwarded to PDSH for parallel execution
 on all nodes.

Selected Nodes:
node0
node2
node1
node4
node7
node5
node6

Nodes were filtered by PBS.

beosh> dmode
beosh> history
   0  pmode
   1  hostname
   2  dmode
   3  status
   4  sleep 10 &
   5  sleep 15 &
   6  sleep 10 &
   7  sleep 20 &
   8  sleep 5 &
   9  jobs
  10  jobs
  11  jobs
  12  ls
  13  pwd
```

```
  14  cd ..
  15  pwd
  16  ls
  17  cd
  18  pwd
  19  ls -alh
  20  ls -alhR | grep mvail | grep rw | grep Jun | sort | sort -r
 | grep K | grep _ | grep c$ | grep beosh
  21  help
  22  status
  23  hostname
  24  hostname
  25  hostname
  26  hostname
  27  hostname
  28  hostname
  29  hostname
  30  hostname
  31  hostname
  32  hostname
  33  hostname
  34  hostname
  35  hostname
  36  hostname
  37  hostname
  38  pmode
  39  hostname
  40  status
  41  dmode
  42  history
beosh> ls -alhR | grep mvail | grep rw | grep Jun | sort | sort -r
 | grep K | grep _ | grep c$ | grep beosh &

[1] 28829

beosh> ls
beosh
beosh_snapshot_06_21.README
beosh_snapshot_06_21.tgz
dcmd.log
dcmdmgr
make_beosh.sh
```

```
my_make_beosh.sh
projectbackups
pvm3
src.tgz
beosh> -rw-rw-r--  1 mvail mvail 9.2K Jun  4 16:22 beosh_list_node.c
-rw-rw-r--  1 mvail mvail 8.9K Jun 15 22:03 beosh_parse.c
-rw-rw-r--  1 mvail mvail 3.7K Jun  4 16:22 beosh_list.c
-rw-rw-r--  1 mvail mvail 3.2K Jun  7 19:04 beosh_error.c
-rw-rw-r--  1 mvail mvail 2.6K Jun  4 16:22 beosh_node.c
-rw-rw-r--  1 mvail mvail 1.9K Jun  4 16:22 beosh_node_node.c
-rw-rw-r--  1 mvail mvail  19K Jun 21 15:41 beosh_util.c
-rw-rw-r--  1 mvail mvail  19K Jun 17 17:08 beosh_builtin.c
-rw-rw-r--  1 mvail mvail  16K Jun 21 15:41 beosh_io.c
-rw-rw-r--  1 mvail mvail  16K Jun 17 17:08 beosh_exec.c

[1] Done ls -alhR | grep mvail | grep rw | grep Jun | sort | sort -r
 | grep K | grep _ | grep c$ | grep beosh  &
beosh>
beosh> cat dcmd.log
[rookery.boisestate.edu] 28671 dcmdmgr for sleep 10  starting
[node2] 9764 dcmdmgr for sleep 15  starting
[node1] 24039 dcmdmgr for sleep 10  starting
[rookery.boisestate.edu] 28671 dcmdmgr for sleep 10  exiting
[node4] 30179 dcmdmgr for sleep 20  starting
[node7] 15971 dcmdmgr for sleep 5  starting
[node1] 24039 dcmdmgr for sleep 10  exiting
[node2] 9764 dcmdmgr for sleep 15  exiting
[node7] 15971 dcmdmgr for sleep 5  exiting
[node4] 30179 dcmdmgr for sleep 20  exiting
[node5] 32468 dcmdmgr for ls starting
[node5] 32468 dcmdmgr for ls exiting
[node6] 6482 dcmdmgr for pwd starting
[node6] 6482 dcmdmgr for pwd exiting
[rookery.boisestate.edu] 28702 dcmdmgr for pwd starting
[rookery.boisestate.edu] 28702 dcmdmgr for pwd exiting
[node2] 9775 dcmdmgr for ls starting
[node2] 9775 dcmdmgr for ls exiting
[node1] 24051 dcmdmgr for pwd starting
[node1] 24051 dcmdmgr for pwd exiting
[node4] 30191 dcmdmgr for ls -alh starting
[node4] 30191 dcmdmgr for ls -alh exiting
[node7] 15996 dcmdmgr for  grep beosh starting
```

```
[node7] 15999 dcmdmgr for  grep c$  starting
[node5] 32488 dcmdmgr for  grep _  starting
[node5] 32492 dcmdmgr for  grep K  starting
[node6] 6504 dcmdmgr for  sort -r  starting
[node6] 6507 dcmdmgr for  sort  starting
[rookery.boisestate.edu] 28736 dcmdmgr for  grep Jun  starting
[node2] 9795 dcmdmgr for  grep rw  starting
[node1] 24069 dcmdmgr for  grep mvail  starting
[node4] 30209 dcmdmgr for ls -alhR  starting
[node4] 30209 dcmdmgr for ls -alhR  exiting
[node1] 24069 dcmdmgr for  grep mvail  exiting
[node2] 9795 dcmdmgr for  grep rw  exiting
[rookery.boisestate.edu] 28736 dcmdmgr for  grep Jun  exiting
[node6] 6507 dcmdmgr for  sort  exiting
[node6] 6504 dcmdmgr for  sort -r  exiting
[node5] 32492 dcmdmgr for  grep K  exiting
[node5] 32488 dcmdmgr for  grep _  exiting
[node7] 15999 dcmdmgr for  grep c$  exiting
[node7] 15996 dcmdmgr for  grep beosh exiting
[rookery.boisestate.edu] 28747 dcmdmgr for hostname starting
[rookery.boisestate.edu] 28747 dcmdmgr for hostname exiting
[node2] 9803 dcmdmgr for hostname starting
[node2] 9803 dcmdmgr for hostname exiting
[node1] 24077 dcmdmgr for hostname starting
[node1] 24077 dcmdmgr for hostname exiting
[node4] 30217 dcmdmgr for hostname starting
[node4] 30217 dcmdmgr for hostname exiting
[node7] 16009 dcmdmgr for hostname starting
[node7] 16009 dcmdmgr for hostname exiting
[node5] 32502 dcmdmgr for hostname starting
[node5] 32502 dcmdmgr for hostname exiting
[node6] 6516 dcmdmgr for hostname starting
[node6] 6516 dcmdmgr for hostname exiting
[rookery.boisestate.edu] 28774 dcmdmgr for hostname starting
[rookery.boisestate.edu] 28774 dcmdmgr for hostname exiting
[node2] 9808 dcmdmgr for hostname starting
[node2] 9808 dcmdmgr for hostname exiting
[node1] 24083 dcmdmgr for hostname starting
[node1] 24083 dcmdmgr for hostname exiting
[node4] 30223 dcmdmgr for hostname starting
[node4] 30223 dcmdmgr for hostname exiting
[node7] 16014 dcmdmgr for hostname starting
```

```
[node7] 16014 dcmdmgr for hostname exiting
[node5] 32508 dcmdmgr for hostname starting
[node5] 32508 dcmdmgr for hostname exiting
[node6] 6522 dcmdmgr for hostname starting
[node6] 6522 dcmdmgr for hostname exiting
[rookery.boisestate.edu] 28800 dcmdmgr for hostname starting
[rookery.boisestate.edu] 28800 dcmdmgr for hostname exiting
[node2] 9829 dcmdmgr for  grep beosh  starting
[node2] 9831 dcmdmgr for  grep c$  starting
[node1] 24103 dcmdmgr for  grep _  starting
[node1] 24106 dcmdmgr for  grep K  starting
[node4] 30243 dcmdmgr for  sort -r  starting
[node4] 30246 dcmdmgr for  sort  starting
[node7] 16035 dcmdmgr for  grep Jun  starting
[node5] 32528 dcmdmgr for  grep rw  starting
[node7] 16039 dcmdmgr for ls starting
[node7] 16039 dcmdmgr for ls exiting
[rookery.boisestate.edu] 28835 dcmdmgr for ls -alhR  starting
[node6] 6542 dcmdmgr for  grep mvail  starting
[rookery.boisestate.edu] 28835 dcmdmgr for ls -alhR  exiting
[node6] 6542 dcmdmgr for  grep mvail  exiting
[node5] 32528 dcmdmgr for  grep rw  exiting
[node7] 16035 dcmdmgr for  grep Jun  exiting
[node4] 30246 dcmdmgr for  sort  exiting
[node4] 30243 dcmdmgr for  sort -r  exiting
[node1] 24106 dcmdmgr for  grep K  exiting
[node1] 24103 dcmdmgr for  grep _  exiting
[node2] 9831 dcmdmgr for  grep c$  exiting
[node2] 9829 dcmdmgr for  grep beosh  exiting
[node5] 32534 dcmdmgr for cat dcmd.log starting
beosh> ls -alhR | grep mvail | grep rw | grep Jun | sort | sort -r
 | grep K | grep _ | grep c$ | grep beosh > output
beosh> ls
beosh
beosh_snapshot_06_21.README
beosh_snapshot_06_21.tgz
dcmd.log
dcmdmgr
make_beosh.sh
my_make_beosh.sh
output
projectbackups
```

```
pvm3
src.tgz
beosh> cat output
-rw-rw-r--  1 mvail mvail 9.2K Jun  4 16:22 beosh_list_node.c
-rw-rw-r--  1 mvail mvail 8.9K Jun 15 22:03 beosh_parse.c
-rw-rw-r--  1 mvail mvail 3.7K Jun  4 16:22 beosh_list.c
-rw-rw-r--  1 mvail mvail 3.2K Jun  7 19:04 beosh_error.c
-rw-rw-r--  1 mvail mvail 2.6K Jun  4 16:22 beosh_node.c
-rw-rw-r--  1 mvail mvail 1.9K Jun  4 16:22 beosh_node_node.c
-rw-rw-r--  1 mvail mvail  19K Jun 21 15:41 beosh_util.c
-rw-rw-r--  1 mvail mvail  19K Jun 17 17:08 beosh_builtin.c
-rw-rw-r--  1 mvail mvail  16K Jun 21 15:41 beosh_io.c
-rw-rw-r--  1 mvail mvail  16K Jun 17 17:08 beosh_exec.c
beosh> rm output
beosh> ls
beosh
beosh_snapshot_06_21.README
beosh_snapshot_06_21.tgz
dcmd.log
dcmdmgr
make_beosh.sh
my_make_beosh.sh
projectbackups
pvm3
src.tgz
beosh> sleep 20 &

[1] 28879

beosh> sleep 15 &

[2] 28882

beosh> sleep 10 &

[3] 28889

beosh> sleep 5 &

[4] 28893

beosh> jobs
```

```
[1] Running sleep 20  &
[2] Running sleep 15  &
[3] Running sleep 10  &
[4] Running sleep 5  &
beosh>
beosh> jobs
[1] Running sleep 20  &
[2] Running sleep 15  &
[3] Running sleep 10  &
[4] Running sleep 5  &
beosh> jobs
[1] Running sleep 20  &
[2] Running sleep 15  &
[3] Running sleep 10  &
[4] Done sleep 5  &
beosh> jobs
[1] Running sleep 20  &
[2] Running sleep 15  &
[3] Done sleep 10  &
beosh> jobs
[1] Running sleep 20  &
[2] Done sleep 15  &
beosh> jobs
[1] Done sleep 20  &
beosh> jobs
beosh> sleep 20
[1] Stopped sleep 20

beosh> bg
[1] sleep 20 &
beosh> jobs
[1] Running sleep 20 &
beosh> jobs
[1] Running sleep 20 &
beosh> jobs
[1] Running sleep 20 &
beosh> jobs
[1] Running sleep 20 &
beosh> jobs
[1] Done sleep 20 &
beosh> jobs
beosh> sleep 10
```

```
[1] Killed sleep 10 &
beosh> jobs
beosh> sleep 10 &

[1] 28919

beosh>

No foreground process

beosh> bg

No current stopped job!

beosh> fg

No current stopped job!

beosh> jobs
[1] Done sleep 10  &
beosh> sleep 10
[1] Stopped sleep 10

beosh> jobs
[1] Stopped sleep 10
beosh> bg
[1] sleep 10 &
beosh> jobs
[1] Running sleep 10 &
beosh> fg

No current stopped job!

beosh> fg 1
[1] Killed sleep 10 &
Job did not confirm continued status.  Job may or may not be running.
beosh> jobs
beosh> sleep 10
[1] Stopped sleep 10

beosh> bg
[1] sleep 10 &
```

```
beosh> fg 1
sleep 10
beosh> jobs
beosh> sleep 10 &

[1] 28941

beosh> jobs
[1] Running sleep 10  &
beosh> fg 1
sleep 10
beosh> exit

Et tu, Brutus?

[mvail@rookery PBS ~]:qstat -n

rookery.boisestate.edu:
                                  Req'd   Elap
Job ID     Username Queue    NDS Time  S Time
---------- -------- -------- --- ----- - -----
48.rookery mvail    default  7   00:30 R   --
   node0/0+node7/0+node6/0+node5/0+node4/0+node2/0+node1/0
[mvail@rookery PBS ~]:qdel 48.rookery

qsub: job 48.rookery.boisestate.edu completed
[mvail@rookery ~]$ qstat -n
[mvail@rookery ~]$ beosh -a
Reserve nodes through PBS and try again.

[mvail@rookery ~]$ beosh -a -o -x node3
beosh> date
Tue Jul  4 16:33:03 MDT 2006
beosh> hostname
node4
beosh> hostname
node1
beosh> exit

Et tu, Brutus?

[mvail@rookery ~]$ beosh -a -o -x node3 -p hostname
```

```
node0: rookery.boisestate.edu
node5: node5
node6: node6
node4: node4
node1: node1
node7: node7
node2: node2
[mvail@rookery ~]$ beosh -a -o -x node3 hostname
rookery.boisestate.edu
[mvail@rookery ~]$ beosh -ao beosh -w node4 -o hostname
node4
[mvail@rookery ~]$ beosh -w node4 -o beosh -w node5 -o hostname
node5
[mvail@rookery ~]$ beosh -aop beosh -ao hostname
node5: node5
node7: node7
node4: node4
node2: node2
node6: node6
node1: node1
node0: rookery.boisestate.edu
[mvail@rookery ~]$ beosh -aop beosh -aop hostname
node7: node7: node7
node7: node6: node6
node7: node4: node4
node7: node5: node5
node5: node5: node5
node7: node0: rookery.boisestate.edu
node5: node4: node4
node7: node1: node1
node7: node2: node2
node2: node2: node2
node1: node0: rookery.boisestate.edu
node1: node1: node1
node1: node2: node2
node1: node6: node6
node2: node1: node1
node2: node7: node7
node2: node5: node5
node2: node4: node4
node2: node0: rookery.boisestate.edu
node1: node5: node5
```

```
node1: node4: node4
node5: node6: node6
node5: node1: node1
node5: node7: node7
node5: node2: node2
node1: node7: node7
node4: node6: node6
node4: node5: node5
node2: node6: node6
node4: node4: node4
node4: node1: node1
node4: node2: node2
node4: node7: node7
node4: node0: rookery.boisestate.edu
node5: node0: rookery.boisestate.edu
node6: node6: node6
node6: node5: node5
node6: node4: node4
node6: node1: node1
node6: node2: node2
node6: node7: node7
node6: node0: rookery.boisestate.edu
node0: node0: rookery.boisestate.edu
node0: node6: node6
node0: node5: node5
node0: node7: node7
node0: node4: node4
node0: node2: node2
node0: node1: node1
[mvail@rookery ~]$
```

# Chapter 7

# IMPROVING BEOSH

## 7.1 Conversion to SSH and Persistent Command Sockets

The most critical need for `beosh` is conversion from `rsh` to a more scalable remote protocol like `ssh`. With `rsh`, reserved ports are used to start each `dcmdmgr` for each client creating competition between `beosh` users for resources. On the target clusters, reserved ports are also used by other services such as Network File System (NFS). Other applications such as `pdsh` account for `rsh` limitations with a sliding window approach to limit the number of restricted ports in simultaneous use. The `beosh` client, however, needs to be able to start an arbitrary number of remote processes at once and keep the remote connections open indefinitely, making `rsh` an untenable long-term solution.

`Beosh` used `rsh` during initial development because the Tux test cluster was not set up for passwordless remote command execution using `ssh`. It was expected at the time that swapping `ssh` into the place of `rsh` would be a negligible effort. Unfortunately, the switch was not so simple. Client management of `dcmdmgr`s was designed around `rsh` behavior as described in Section 3.2.1. A child `rsh` process exits when

its remote process exits, allowing the parent client to recognize when a `dcmdmgr` exits. Child `ssh` processes do not exit. Likewise, when the client kills a child `rsh` process, the remote process is also killed allowing the client to clean up during a refresh or when exiting. Killing a local `ssh` process does not kill the remote process.

Converting `beosh` to use `ssh` requires a change to the design by which clients monitor and interact with their remote `dcmdmgr`s. To maximize scalability, the client was designed to have no permanent communication connections to `dcmdmgr`s. New sockets are created for each new command as in a stateless client-server architecture. If this basic design were retained, a ping command would suffice for verifying a waiting remote process, but would not distinguish between a hung process, a busy process, or a dead process. An alternative to continual pinging is to ping and refresh all remote processes only when a connection attempt fails, trading a constant level of overhead for generally faster response interrupted by occasional long waits. However, if the client does not recognize that a `dcmdmgr` is unresponsive until part way through distributing commands for a pipeline and a new port has to be chosen, any `dcmdmgr`s that already received assignments will be left in a hung state, waiting for pipeline connections that will never arrive. For either of these options, establishment and closure of socket connections for every ping of every `dcmdmgr` would create excessive overhead.

Rather than create numerous temporary sockets, a permanent command socket should be established between the client and each `dcmdmgr` when the `dcmdmgr` is

started. When a `dcmdmgr` dies, the socket `EOF` alerts the client that it has died. Until then, all commands and job assignments would be delivered to the `dcmdmgr` over the socket. Because a hung `dcmdmgr` cannot respond to a kill command, a kill signal from the client must be delivered through a separate remote process. This design was rejected early in the project because it ties up more ports on the client's node and is, therefore, less scalable than the stateless design. However, maintaining permanent sockets provides performance and security benefits over the stateless design. With establishment of a permanent command socket, there is no need for authentication before accepting commands. There is also no delay in creating and closing socket connections for every command and job assignment making `beosh` much more responsive.

## 7.2 Other Opportunities For Improvement

### 7.2.1 Improving SSI

`Beosh` does not not allow users to execute interactive processes such as `vi` remotely. This includes commands that return a prompt for a password such as `su`. Though this limitation is shared by other cluster shells including `pdsh`, it still disrupts the SSI user experience and prevents a user from using `beosh` as a default, general-purpose shell.

Another disruption to the SSI user experience is lack of synchronization of environment variables. Though the current working directory is synchronized across all

session nodes, environment variables are not. A simple workaround until such capability is added to `beosh` is to set any critical environment variables on all session nodes using `beosh` in parallel mode or `pdsh` directly prior to executing any jobs that rely on a particular environment variable.

### 7.2.2 Job Assignment

While it is expected that `beosh` would be used most often as an SSI shell shielding users from having to be aware of the cluster behind the prompt, some users will want more explicit control of where commands execute. Explicit command assignment is actually not possible at this time, though such capability could certainly be added.

Uniform job distribution through the balanced round-robin-based algorithm does not take into account potential hardware differences between nodes of a cluster. Though not currently implemented, `beosh` could be modified to recognize repeated instances of particular nodes when parsing the node list and assign additional commands to those nodes.

A weakness of round-robin-based assignment is the potential for repeatedly assigning demanding processes to the same node. Batch scripts running many variations of the same pipelined job, for example, may continually wrap around the node list and place the most demanding processes on the same nodes. A simple solution to this problem while retaining the uniform process distribution of round robin is to randomly reorder the `beosh` node list after each iteration through all of the session nodes.

### 7.2.3 Security

Security is basic. The client and backend command managers are user-level processes, so users can only execute and access files with their own permissions. However, the simple authentication of command connections is based on the assumption that the first command connection to a newly-started `dcmdmgr` will be from the legitimate client. Implementing persistent command sockets as described in Section 7.1 would remove the need to check the source of every command, but a more trustworthy authentication scheme is needed for establishing those sockets.

There is also no encryption of network traffic through pipelines allowing easy viewing of other users' traffic. The naive assumption is that most clusters are behind a substantial firewall and access is restricted to trustworthy users. As with all such assumptions, unfortunately, it is almost certainly wrong. However, encryption of data through secure socket layer (SSL) or some other means could negatively impact performance. Performance impacts should be considered if data encryption is to be implemented.

# Chapter 8

# CONCLUSIONS

## 8.1 Toward a Single System Image (SSI) Distributed Shell

`Beosh` provides a solid foundation for a production-quality, SSI cluster shell. As presented here, it is suitable for use in a small cluster environment where users are aware of the exposure of their data over unencrypted sockets and where there is little competition for reserved ports required by `rsh`. `Beosh` can be configured to limit users to nodes reserved through the Portable Batch System (PBS).

`Beosh` achieves distribution of individual commands and command pipelines across available nodes without the user needing to specify where any command goes. `Beosh` provides standard job control capabilities for distributed jobs including starting a job in foreground or background, stopping a foreground job, and restarting a stopped job in the foreground or background. The user's current working directory is also updated across all session `dcmdmgr`s when changed in the client.

## 8.2 Code Reuse and Maintainability

Use of `pdsh` for hostname list expansion and parallel command execution avoids reinventing existing functionality and relieves `beosh` managers from maintaining parallel functionality. Options when starting `beosh` follow `pdsh` syntax for easy migration between shells.

## 8.3 Portability

In addition to standard system calls, `beosh` uses only utilities commonly found or widely available on Beowulf clusters including `rsh` or `ssh` (for remotely starting backend command managers), `awk` (for selecting hostnames from `pdsh` output), and `pdsh`. Standard sockets and pipes are used for all interprocess communication.

`Beosh` has yet to be tested on a non-Linux cluster, but it has been built and tested without requiring any changes on several clusters of different sizes and with differing hardware, kernels, and compiler versions.

## 8.4 Performance and Overhead

Load balancing is through even distribution of pipelined commands across selected nodes and a variation on round-robin assignment of the first node for each job. Implementation of self-assembling communications pipelines through backend command managers avoids communications bottlenecks and improves scalability. Though sub-

jective performance seems high once a job is running, the overhead of starting up backend managers, distributing command assignments, and establishing pipelines is definitely noticeable for short-duration jobs and can detract from interactive sessions.

Each client session manages a single set of backend managers for all jobs, minimizing required resources for overhead. Because backend managers only run for the duration of a client session, resources are only used while needed.

Port selection using a hash based on client `pid` and the ability to choose a different port when the selected port is already in use allows multiple users and sessions to use nodes concurrently without any need to explicitly coordinate between them.

# REFERENCES


[1] R.W. Clarke and B.T.B. Lee. Cluster Operating Systems. World Wide Web. 2000. http://www.buyya.com/csc433/ClusterOS.pdf

[2] L. Fried and R. Tibbetts. Distributed Command Line Interface. Dec 2001. http://www.pdos.lcs.mit.edu/6.824-2001/projects/paper-3.ps

[3] J. Garlick. Pdsh parallel distributed shell. World Wide Web. Feb 2003. http://www.llnl.gov/Linux/pdsh/index.html

[4] D.P. Ghormley, D. Petrou, S.H. Rodrigues, A.M. Vahdat and T.E. Anderson. GLUnix: a Global Layer Unix for a Network of Workstations. Software Practice and Experience, Vol 28(9), 1998:929-961, http://now.cs.berkeley.edu/Glunix/glunix.html

[5] Edward A. Lee. *The Problem With Threads.* EEECS Department, University of California, Berkeley. Technical Report No. USB/EEECS-2006-1. Jan 2006. http://www.eeecs.berkeley.edu/Pubs/TechRpts/2006/EEECS-2006-1.html

[6] MOSIX. World Wide Web. http://www.mosix.org

[7] J. Rappleye. dsh distributed shell. World Wide Web. Feb 2003. http://www.ccr.buffalo.edu/rappleye/dsh.html

[8] C.M. Tan, C.P. Tan and W.F. Wong. Shell over a Cluster (SHOC): Towards Achieving Single System Image via the Shell. In 2002 IEEE International Conference on Cluster Computing, pp. 28-34. IEEE, September 2002. http://www.comp.nus.edu.sg/~wongwf/papers/cluster2002.pdf

[9] F. Wheeler. dish distributed shell. World Wide Web. Jun 2002. http://www.cipr.rpi.edu/~wheeler/dish/index.html

[10] Mason E. Vail. BEOSH and pbsnodefile project page. World Wide Web. 2006. http://cs.boisestate.edu/~amit/research/beosh

# Appendix A

# BEOSH MAN PAGE

BEOSH(1) BEOSH(1)

NAME

beosh - Beowulf cluster shell

SYNOPSIS

beosh [options]... [command]

DESCRIPTION

beosh is a Single System Image (SSI) cluster shell enabling distribution of non-interactive commands across available nodes without having to specify exactly where any of them should execute. Standard job control is available for jobs distributed this way. If parallel execution is specified, commands are forwarded to pdsh(1).

If a remote command is not specified on the command line, beosh runs interactively, prompting for commands and executing them when terminated with a carriage return.

pdsh(1) using the machines module is required as a dependency for beosh for nodelist expansion and parallel command execution. beosh node option syntax follows that of pdsh(1) to ease user transition between shells.

If configured for Portable Batch System (PBS) support, beosh will filter the nodelist returned by pdsh(1) to remove nodes not currently reserved through PBS. beosh also supports PBS filtering if pdsh(1) uses the pbsnodefile module.

OPTIONS

Target nodelist options use pdsh(1) syntax. To these options, beosh adds an option for parallel execution and for overriding PBS filtering if PBS filtering is configured.

Target nodelist options

-w node,node,...
: Target the specified list of nodes. Do not use with -a. No spaces are allowed in the comma-separated list. The node list may contain nodelist expressions of the form ‘‘node[1-5,7]’’.
For more information about the nodelist format, see the HOSTLIST EXPRESSIONS below.

-a
: Target all nodes from machines file.

-x node,node,...
: Exclude the specified nodes. May be specified in conjunction with other target node list options such as -a and -w.
Hostlists may also be specified to the -x option (see the HOSTLIST EXPRESSIONS below).

Parallel execution option

-p
: Rather than distribute commands, execute commands in parallel on all specified nodes using pdsh(1). In parallel mode, job control is not available.

PBS filtering option

-o
: Override PBS filtering if PBS filtering is configured for beosh or pdsh(1) uses the pbsnodefile module.

Other options

-h
: Output usage menu and quit.

HOSTLIST EXPRESSIONS

As noted in sections above, beosh follows pdsh(1) syntax for nodelist selection and accepts lists of nodes with the general form: prefix[n-m,l-k,...], where n < m and l < k, etc., as an alternative to explicit lists of nodes. This form should not be confused with regular expression character classes (also denoted by ‘‘[]’’). For example, foo[19] does not represent an expression matching foo1 or foo9, but rather represents the degenerate nodelist: foo19.

The nodelist syntax is meant only as a convenience on clusters with a "prefixNNN" naming convention and specification of ranges should not be considered necessary -- this foo1,foo9 could be specified as such, or by the nodelist foo[1,9].

Some examples of usage assuming PBS filtering follow:

Run command using reserved nodes in foo01,foo02,...,foo05

```
beosh -w foo[01-05] command
```

Run command in parallel on foo7,foo9,foo10 overriding PBS filtering

```
beosh -w foo[7,9-10] -p -o command
```

Run command using foo0,foo4,foo5

```
beosh -w foo[0-5] -x foo[1-3] -o command
```

Run command using all reserved nodes

```
beosh -a command
```

As a reminder to the reader, some shells will interpret brackets (’[’ and ’]’) for pattern matching. Depending on your shell, it may be necessary to enclose ranged lists within quotes. For example, in tcsh, the first example above should be executed as:

```
beosh -w "foo[01-05]" command
```

ORIGIN

A Master's project by Mason Vail <masonvail@mail.boisestate.edu> at Boise State University, written for use on student and research clusters.

LIMITATIONS

beosh currently uses rsh(1) to start backend distributed command managers (dcmdmgrs) which causes a potential scalability problem due to rsh(1) use of reserved ports.

Hostlist parsing assumes numerical part of hostname is at the end only, e.g. specifying foo[0-5]bar will not work.

Though current working directory is synchronized across session nodes, environment variables are not. Prior to executing commands with beosh depending on particular environment variables, set them on all nodes using beosh in parallel mode or with pdsh(1).

beosh does not allow execution of interactive commands or commands that will result in prompts such as password authentication.

FILES

SEE ALSO

pdsh(1), pbs(1B), rsh(1)

# Appendix B

# SUMMARY OF BUILT-IN COMMANDS

**help** display built-in commands

**exit** quit beosh

**logout** quit beosh

**cd** $< path >$ change directory if in distributed mode

**jobs** list current status of stopped or background jobs

**fg** $< jobnumber >$ run last stopped job or numbered background job in the foreground

**bg** $< jobnumber >$ run last stopped job or numbered background job in the background

**history** $< n >$ show all commands entered in the session or only the last n commands

**dmode** switch to distributed mode

**pmode** switch to parallel mode

**status** display information about current beosh settings and nodes

**version** display current beosh version

**Ctrl-c** kill foreground job (`SIGINT`)

**Ctrl-z** stop foreground job (`SIGTSTP`)

# Appendix C

# BUILD AND TEST ENVIRONMENT

## C.1 Laptop of Joy and Happiness

Coding was done on a beloved Dell Inspiron 9200 with all the hardware goodies.

- 2GHz Pentium M, 1GB RAM
- Fedora Core 3, kernel 2.6.12
- gcc version 3.4.4
- KDevelop version 3.2.2

## C.2 Target Clusters

The following Beowulf clusters at Boise State University are the "target clusters" referred to throughout the report.

### C.2.1 Beowulf

Beowulf is the main research cluster for computer science at Boise State University with a dual-processor head node and 60 dual-processor nodes.

- private gigabit ethernet
- head node: dual 2.4GHz Intel Xeon processors, 4GB RAM
- other nodes: dual 2.4GHz Intel Xeon processors, 1GB RAM
- Fedora Core 3, kernel 2.6.12
- gcc version 3.4.3

### C.2.2 Tux

Tux was a test cluster with 3 nodes isolated from the main Beowulf cluster. Most development testing was done on Tux until it was effectively reduced to a single node and became somewhat less than useful for testing a cluster shell. One node was removed from the cluster shortly after development began for use as the NFS manager for the main cluster. Another node suffered from the slow, sad degradation of its harddrive rendering it unreliable at best. Shortly before the project ended, the remaining node was scavenged for another student project and Tux was no more.

- private gigabit ethernet
- all nodes: dual 2.4GHz Intel Xeon processors, 2GB RAM
- Fedora Core 3, kernel 2.6.11
- gcc version 3.4.3

### C.2.3 Rookery

Rookery is an 8-node student research cluster composed of assorted hardware. After Tux, development testing moved to Rookery. It was extremely gratifying to see the code compile and run on Rookery without warning or error despite it using a different compiler than on the development laptop or on Tux.

- private gigabit ethernet
- head node: 1.7GHz Intel Celeron, 1GB RAM
- nodes 1-3: 1.7GHz Intel Celeron, 512MB RAM
- nodes 4-7: 1.2GHz Intel Pentium 3, 512MB RAM
- Fedora Core 4, kernel 2.6.14
- gcc version 4.0.1

### C.2.4 Onyx

Onyx is the main computer science student Linux lab, but it is also a teaching and student research cluster with a dual-processor head node and 32 nodes.

- private gigabit ethernet
- head node: dual 2.4GHz Intel Xeon processors, 3GB RAM

- nodes 01-05: 1.4GHz AMD Duron processors, 512MB RAM
- nodes 06-32: 2.8GHz Intel Pentium 4 w/HT processors, 1GB RAM
- kernel 2.4.32 #4 SMP
- gcc version 3.3.2

# Appendix D

# PROJECT MANAGEMENT

## D.1 Source Control

For source control I used Subversion 1.1.4 with the BerkleyDB repository. The project repository was originally placed on Beowulf where it would be safely backed up nightly. Combination of a sequence of cluster crashes and the unintentional updating of BerkleyDB on Beowulf as a dependency to another package resulted in corruption and loss of the original repository after several weeks of work. I decided to create the replacement repository on my laptop where I am the only user from local copies of project files, freeze any and all updates to my laptop for the remainder of the project (baring any more catastrophic events), and keep two other backup copies of the repository on tux and onyx in case of damage to the original. Thumbs up to Subversion, but thumbs down to using BerkleyDB for the repository. I will use flat files for future projects. I will also not blindly trust automatic backups on a research system with many users.

## D.2 Where to get Beosh

Beosh was developed under the GNU General Public License version 2 and is available for download from Boise State University.

`http://cs.boisestate.edu/~amit/research/beosh`

Future project development will be managed by Amit Jain at Boise State University.

# Appendix E

# CLOSING QUANDARY

According to Edward A. Lee [5]:

> . . . a folk definition of insanity is to do the same thing over and over again and to expect the results to be different. By this definition, we in fact require that programmers of multithreaded programs be insane. Were they sane, they could not understand their programs.

My dilemma is this - If I succeed in convincing those who require convincing that `beosh` does, in fact, work and, further, that I actually understand how and why it works, have I actually succeeded in arguing for my own insanity, thus proving that I and my work cannot possibly be approved? On the other hand, would I, by disavowing understanding of how it works and even denying that I am sure that it does work, defend my place as a rational, sane member of society (the sort of person I would think would be deserving of approval) but forfeit credit for the work I've done?

These are the sorts of issues that have occupied a great deal of the time I should have been using to figure out what actually does and does not work and deciding how I feel about that. In the end, I am afraid tergiversation may be my only option and I hope the reader will decide in my favor, whatever that means.